WinDriver™ USB User's Manual

Jungo Connectivity Ltd.

Version 11.5.0

WinDriver™ USB User's Manual

Copyright © 2014 Jungo Connectivity Ltd. All Rights Reserved

Information in this document is subject to change without notice. The software described in this document is furnished under a license agreement. The software may be used, copied or distributed only in accordance with that agreement. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or any means, electronically or mechanically, including photocopying and recording for any purpose without the written permission of Jungo Connectivity Ltd.

Brand and product names mentioned in this document are trademarks of their respective owners and are used here only for identification purposes.

Table of Contents

1. WinDriver Overview 1
    1.1. Introduction to WinDriver 1
    1.2. Background 2
        1.2.1. The Challenge 2
        1.2.2. The WinDriver Solution 2
    1.3. Conclusion 3
    1.4. WinDriver Benefits 3
    1.5. WinDriver Architecture 4
    1.6. What Platforms Does WinDriver Support? 5
    1.7. Limitations of the Different Evaluation Versions 5
    1.8. How Do I Develop My Driver with WinDriver? 6
        1.8.1. On Windows and Linux 6
        1.8.2. On Windows CE 6
    1.9. What Does the WinDriver Toolkit Include? 6
        1.9.1. WinDriver Modules 7
        1.9.2. Utilities 7
        1.9.3. Samples 8
    1.10. Can I Distribute the Driver Created with WinDriver? 8
2. Understanding Device Drivers 9
    2.1. Device Driver Overview 9
    2.2. Classification of Drivers According to Functionality 9
        2.2.1. Monolithic Drivers 9
        2.2.2. Layered Drivers 10
        2.2.3. Miniport Drivers 11
    2.3. Classification of Drivers According to Operating Systems 12
        2.3.1. WDM Drivers 12
        2.3.2. Unix Device Drivers 13
        2.3.3. Linux Device Drivers 13
    2.4. The Entry Point of the Driver 13
    2.5. Associating the Hardware with the Driver 14
    2.6. Communicating with Drivers 14
3. WinDriver USB Overview 15
    3.1. Introduction to USB 15
    3.2. WinDriver USB Benefits 16
    3.3. USB Components 16
    3.4. Data Flow in USB Devices 17
    3.5. USB Data Exchange 18
    3.6. USB Data Transfer Types 19
        3.6.1. Control Transfer 20
        3.6.2. Isochronous Transfer 20
        3.6.3. Interrupt Transfer 20
        3.6.4. Bulk Transfer 21
    3.7. USB Configuration 21
    3.8. WinDriver USB 23
    3.9. WinDriver USB Architecture 24
4. Installing WinDriver 26

4.1. System Requirements 26
4.1.1. Windows System Requirements 26
4.1.2. Windows CE System Requirements 26
4.1.3. Linux System Requirements 27
4.2. WinDriver Installation Process 27
4.2.1. Windows WinDriver Installation Instructions 27
4.2.2. Windows CE WinDriver Installation Instructions 28
4.2.2.1. Installing WinDriver CE when Building New CE-Based Platforms 28
4.2.2.2. Installing WinDriver CE when Developing Applications for Windows CE Computers 29
4.2.2.3. Windows CE Installation Note 30
4.2.3. Linux WinDriver Installation Instructions 30
4.2.3.1. Preparing the System for Installation 30
4.2.3.2. Installation 32
4.2.3.3. Restricting Hardware Access on Linux 34
4.3. Upgrading Your Installation 34
4.4. Checking Your Installation 34
4.4.1. Windows and Linux Installation Check 34
4.4.2. Windows CE Installation Check 35
4.5. Uninstalling WinDriver 35
4.5.1. Windows WinDriver Uninstall Instructions 35
4.5.2. Linux WinDriver Uninstall Instructions 37
5. Using DriverWizard 38
5.1. An Overview 38
5.2. DriverWizard Walkthrough 39
5.2.1. Automatic Code Generation 48
5.2.1.1. Generating the Code 48
5.2.1.2. The Generated USB C Code 48
5.2.2. Compiling the Generated Code 48
5.2.2.1. Windows and Windows CE Compilation 48
5.2.2.2. Linux Compilation 49
5.2.3. Bus Analyzer Integration — Ellisys Visual USB 49
6. Developing a Driver 51
6.1. Using DriverWizard to Build a Device Driver 51
6.2. Writing the Device Driver Without DriverWizard 51
6.2.1. Include the Required WinDriver Files 51
6.2.2. Write Your Code 52
6.2.3. Configure and Build Your Code 53
6.3. Developing Your Driver on Windows CE Platforms 53
7. Debugging Drivers 54
7.1. User-Mode Debugging 54
7.2. Debug Monitor 54
7.2.1. The wddebug_gui Utility 55
7.2.1.1. Running wddebug_gui for a Renamed Driver 57
7.2.2. The wddebug Utility 57
7.2.2.1. Console-Mode wddebug Execution 57
7.2.2.2. Windows CE GUI wddebug Execution 60
8. USB Transfers 62

8.1. Overview 62
8.2. USB Control Transfers 63
8.2.1. USB Control Transfers Overview 63
8.2.1.1. Control Data Exchange 63
8.2.1.2. More About the Control Transfer 63
8.2.1.3. The Setup Packet 64
8.2.1.4. USB Setup Packet Format 64
8.2.1.5. Standard Device Request Codes 65
8.2.1.6. Setup Packet Example 65
8.2.2. Performing Control Transfers with WinDriver 66
8.2.2.1. Control Transfers with DriverWizard 66
8.2.2.2. Control Transfers with WinDriver API 68
8.3. Functional USB Data Transfers 69
8.3.1. Functional USB Data Transfers Overview 69
8.3.2. Single-Blocking Transfers 69
8.3.2.1. Performing Single-Blocking Transfers with WinDriver 69
8.3.3. Streaming Data Transfers 69
8.3.3.1. Performing Streaming with WinDriver 70
9. Dynamically Loading Your Driver 72
9.1. Why Do You Need a Dynamically Loadable Driver? 72
9.2. Windows Dynamic Driver Loading 72
9.2.1. The wdreg Utility 72
9.2.1.1. Overview 73
9.2.2. Dynamically Loading/Unloading windrvr6.sys INF Files 74
9.3. Linux Dynamic Driver Loading 75
9.4. Windows CE Dynamic Driver Loading 75
10. Distributing Your Driver 76
10.1. Getting a Valid License for WinDriver 76
10.2. Windows Driver Distribution 76
10.2.1. Preparing the Distribution Package 77
10.2.2. Installing Your Driver on the Target Computer 77
10.3. Windows CE Driver Distribution 80
10.3.1. Distribution to New Windows CE Platforms 80
10.3.2. Distribution to Windows CE Computers 81
10.4. Linux Driver Distribution 82
10.4.1. Preparing the Distribution Package 82
10.4.1.1. Kernel Module Components 83
10.4.1.2. User-Mode Hardware-Control Application or Shared Object 84
10.4.2. Building and Installing the WinDriver Driver Modules on the Target 85
10.4.3. Installing the User-Mode Hardware-Control Application or Shared Object 86
11. Driver Installation — Advanced Issues 87
11.1. Windows INF Files 87
11.1.1. Why Should I Create an INF File? 87
11.1.2. How Do I Install an INF File When No Driver Exists? 88
11.1.3. How Do I Replace an Existing Driver Using the INF File? 88
11.2. Renaming the WinDriver Kernel Driver 89
11.2.1. Windows Driver Renaming 90

11.2.2. Linux Driver Renaming 92
11.3. Windows Digital Driver Signing and Certification 93
11.3.1. Overview 93
11.3.1.1. Authenticode Driver Signature 94
11.3.1.2. Windows Certification Program 94
11.3.2. Driver Signing and Certification of WinDriver-Based Drivers 95
11.3.2.1. HCK Test Notes 96
11.4. Windows XP Embedded WinDriver Component 97
A. 64-Bit Operating Systems Support 99
A.1. Supported 64-Bit Architectures 99
A.2. Support for 32-Bit Applications on 64-Bit Windows and Linux Platforms 99
A.3. 64-Bit and 32-Bit Data Types 100
B. WinDriver USB Host API Reference 101
B.1. WD_DriverName 101
B.2. WinDriver USB (WDU) Library Overview 102
B.2.1. Calling Sequence for WinDriver USB 102
B.2.2. Upgrading from the WD_xxx USB API to the WDU_xxx API 105
B.3. USB User Callback Functions 107
B.3.1. WDU_ATTACH_CALLBACK 107
B.3.2. WDU_DETACH_CALLBACK 108
B.3.3. WDU_POWER_CHANGE_CALLBACK 109
B.4. USB Functions 109
B.4.1. WDU_Init 110
B.4.2. WDU_SetInterface 111
B.4.3. WDU_GetDeviceAddr 112
B.4.4. WDU_GetDeviceRegistryProperty 113
B.4.5. WDU_GetDeviceInfo 114
B.4.6. WDU_PutDeviceInfo 115
B.4.7. WDU_Uninit 116
B.4.8. Single-Blocking Transfer Functions 117
B.4.8.1. WDU_Transfer 117
B.4.8.2. WDU_HaltTransfer 119
B.4.8.3. WDU_TransferDefaultPipe 120
B.4.8.4. WDU_TransferBulk 120
B.4.8.5. WDU_TransferIsoch 121
B.4.8.6. WDU_TransferInterrupt 121
B.4.9. Streaming Data Transfer Functions 122
B.4.9.1. WDU_StreamOpen 122
B.4.9.2. WDU_StreamStart 124
B.4.9.3. WDU_StreamRead 125
B.4.9.4. WDU_StreamWrite 126
B.4.9.5. WDU_StreamFlush 127
B.4.9.6. WDU_StreamGetStatus 128
B.4.9.7. WDU_StreamStop 129
B.4.9.8. WDU_StreamClose 129
B.4.10. WDU_ResetPipe 130
B.4.11. WDU_ResetDevice 131
B.4.12. WDU_SelectiveSuspend 132

B.4.13. WDU_Wakeup 133
B.4.14. WDU_GetLangIDs 134
B.4.15. WDU_GetStringDesc 135
B.5. USB Data Types 136
B.5.1. WD_DEVICE_REGISTRY_PROPERTY Enumeration 136
B.5.2. USB Structures 138
B.5.2.1. WDU_MATCH_TABLE Structure 140
B.5.2.2. WDU_EVENT_TABLE Structure 140
B.5.2.3. WDU_DEVICE Structure 141
B.5.2.4. WDU_CONFIGURATION Structure 142
B.5.2.5. WDU_INTERFACE Structure 142
B.5.2.6. WDU_ALTERNATE_SETTING Structure 143
B.5.2.7. WDU_DEVICE_DESCRIPTOR Structure 143
B.5.2.8. WDU_CONFIGURATION_DESCRIPTOR Structure 144
B.5.2.9. WDU_INTERFACE_DESCRIPTOR Structure 144
B.5.2.10. WDU_ENDPOINT_DESCRIPTOR Structure 145
B.5.2.11. WDU_PIPE_INFO Structure 145
B.6. General WD_xxx Functions 146
B.6.1. Calling Sequence WinDriver — General Use 146
B.6.2. WD_Open() 147
B.6.3. WD_Version() 148
B.6.4. WD_Close() 149
B.6.5. WD_Debug() 150
B.6.6. WD_DebugAdd() 151
B.6.7. WD_DebugDump() 153
B.6.8. WD_Sleep() 154
B.6.9. WD_License() 155
B.7. User-Mode Utility Functions 157
B.7.1. Stat2Str 157
B.7.2. get_os_type 158
B.7.3. ThreadStart 159
B.7.4. ThreadWait 160
B.7.5. OsEventCreate 161
B.7.6. OsEventClose 162
B.7.7. OsEventWait 163
B.7.8. OsEventSignal 164
B.7.9. OsEventReset 165
B.7.10. OsMutexCreate 166
B.7.11. OsMutexClose 167
B.7.12. OsMutexLock 168
B.7.13. OsMutexUnlock 169
B.7.14. PrintDbgMessage 170
B.7.15. WD_LogStart 171
B.7.16. WD_LogStop 172
B.7.17. WD_LogAdd 172
B.8. WinDriver Status Codes 173
B.8.1. Introduction 173
B.8.2. Status Codes Returned by WinDriver 173

B.8.3. Status Codes Returned by USBD 174
C. Troubleshooting and Support 178
D. Evaluation Version Limitations 179
D.1. Windows WinDriver Evaluation Limitations 179
D.2. Windows CE WinDriver Evaluation Limitations 179
D.3. Linux WinDriver Evaluation Limitations 180
E. Purchasing WinDriver 181
F. Distributing Your Driver — Legal Issues 182
G. Additional Documentation 183

List of Figures

1.1. WinDriver Architecture 4
2.1. Monolithic Drivers 10
2.2. Layered Drivers 11
2.3. Miniport Drivers 12
3.1. USB Endpoints 18
3.2. USB Pipes 19
3.3. Device Descriptors 22
3.4. WinDriver USB Architecture 25
5.1. Create or Open a WinDriver Project 39
5.2. Select Your Device 40
5.3. DriverWizard INF File Information 41
5.4. DriverWizard Multi-Interface INF File Information — Specific Interface 42
5.5. DriverWizard Multi-Interface INF File Information — Composite Device 43
5.6. Select Device Interface 44
5.7. USB Control Transfers 45
5.8. Listen to Pipe 46
5.9. Write to Pipe 46
5.10. Code Generation Options 47
5.11. Ellisys Visual USB Integration 50
7.1. Start Debug Monitor 55
7.2. Debug Options 56
7.3. wddebug Windows CE Start Log Message 60
7.4. wddebug Windows CE Stop Log Message 61
8.1. USB Data Exchange 62
8.2. USB Read and Write 64
8.3. Custom Request 67
8.4. Request List 67
8.5. USB Request Log 68
B.1. WinDriver USB Calling Sequence 104
B.2. WinDriver USB Structures 139
B.3. WinDriver-API Calling Sequence 146

Chapter 1
WinDriver Overview

In this chapter you will explore the uses of WinDriver, and learn the basic steps of creating your driver.

This manual outlines WinDriver's support for USB devices.
WinDriver also supports development for PCI/PCMCIA/CardBus/ISA/EISA/ CompactPCI/PCI Express devices. For detailed information regarding WinDriver's support for these buses, please refer to the WinDriver product page on our web site (http://www.jungo.com/st/products/windriver/) and to the **WinDriver PCI Manual**, which is available online at http://www.jungo.com/st/support/windriver/.

1.1. Introduction to WinDriver

WinDriver is a development toolkit that dramatically simplifies the difficult task of creating device drivers and hardware access applications. WinDriver includes a wizard and code generation features that automatically detect your hardware and generate the driver to access it from your application. The driver and application you develop using WinDriver is source code compatible across all supported operating systems [1.6]. The driver is binary compatible across Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/ XP.

WinDriver provides a complete solution for creating high-performance drivers.

Don't let the size of this manual fool you. WinDriver makes developing device drivers an easy task that takes hours instead of months. Most of this manual deals with the features that WinDriver offers to the advanced user. However, most developers will find that reading this chapter and glancing through the DriverWizard and function reference chapters is all they need to successfully write their driver.

WinDriver supports development for all USB chipsets.

Visit Jungo's web site at http://www.jungo.com for the latest news about WinDriver and other driver development tools that Jungo offers.

1.2. Background

1.2.1. The Challenge

In protected operating systems such as Windows and Linux, a programmer cannot access hardware directly from the application level (user mode), where development work is usually done. Hardware can only be accessed from within the operating system itself (kernel mode or Ring-0), utilizing software modules called device drivers. In order to access a custom hardware device from the application level, a programmer must do the following:

- Learn the internals of the operating system he is working on.
- Learn how to write a device driver.
- Learn new tools for developing/debugging in kernel mode (WDK, ETK, DDI/DKI).
- Write the kernel-mode device driver that does the basic hardware input/output.
- Write the application in user mode that accesses the hardware through the device driver written in kernel mode.
- Repeat the first four steps for each new operating system on which the code should run.

1.2.2. The WinDriver Solution

- **Easy Development**: WinDriver enables Windows, Windows CE, and Linux programmers to create USB based device drivers in an extremely short time. WinDriver allows you to create your driver in the familiar user-mode environment, using MS Visual Studio, MS eMbedded Visual C++, MS Platform Builder C++, GCC, Windows GCC, or any other appropriate compiler or development environment. You do not need to have any device driver knowledge, nor do you have to be familiar with operating system internals, kernel programming, the WDK, ETK or DDI/DKI.
- **Cross Platform**: The driver created with WinDriver will run on Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP, Embedded Windows 8.1/8/7/XP, Windows CE (a.k.a. Windows Embedded Compact) 4.x–7.x (including Windows Mobile), and Linux. In other words — write it once, run it on many platforms.
- **Friendly Wizards**: DriverWizard (included) is a graphical diagnostics tool that lets you view the device's resources and test the communication with the hardware with just a few mouse clicks, before writing a single line of code. Once the device is operating to your satisfaction, DriverWizard creates the skeletal driver source code, giving access functions to all the resources on the hardware.
- **Kernel-Mode Performance**: WinDriver's API is optimized for performance.

1.3. Conclusion

Using WinDriver, a developer need only do the following to create an application that accesses the custom hardware:

- Start DriverWizard and detect the hardware and its resources.
- Automatically generate the device driver code from within DriverWizard, or use one of the WinDriver samples as the basis for the application.
- Modify the user-mode application, as needed, using the generated/sample functions to implement the desired functionality for your application.

Your hardware access application will run on all the supported platforms [1.6] — just recompile the code for the target platform. The code is binary compatible across Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP platforms; there is no need to rebuild the code when porting it across binary-compatible platforms.

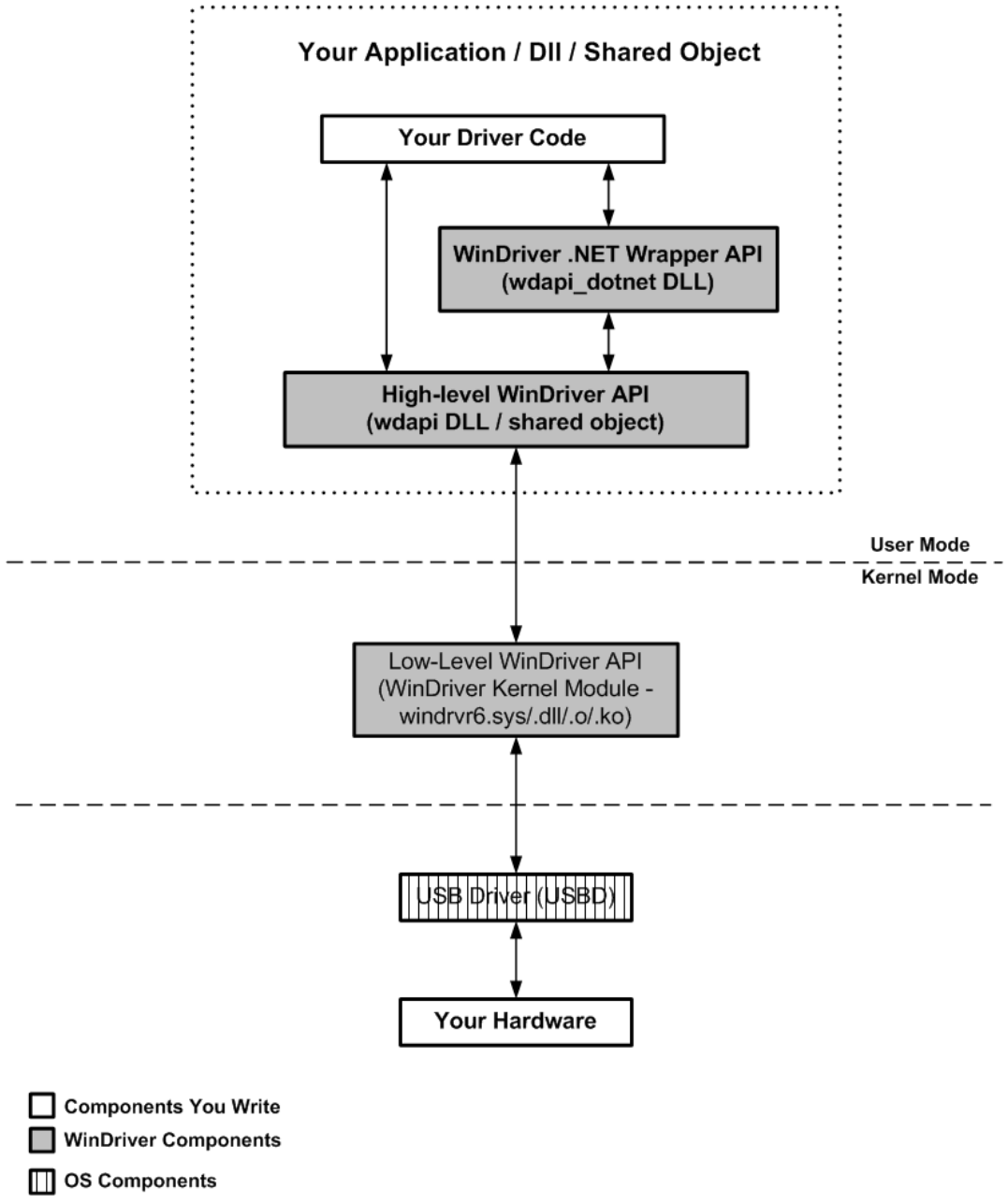
1.4. WinDriver Benefits

- Easy user-mode driver development.
- Friendly DriverWizard allows hardware diagnostics without writing a single line of code.
- Automatically generates the driver code for the project in C, Visual Basic .NET, or C#.
- Supports any USB device, regardless of manufacturer.
- Applications are binary compatible across Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP.
- Applications are source code compatible across all supported operating systems — Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP, Embedded Windows 8.1/8/7/XP, Windows CE (a.k.a. Windows Embedded Compact) 4.x–7.x (including Windows Mobile), and Linux.
- Can be used with common development environments, including MS Visual Studio, MS eMbedded Visual C++, MS Platform Builder C++, GCC, Windows GCC, or any other appropriate compiler/environment.
- No WDK, ETK, DDI or any system-level programming knowledge required.
- Supports multiple CPUs.
- Includes dynamic driver loader.
- Comprehensive documentation and help files.

- Detailed examples in C, Visual Basic .NET, or C#.
- WinDriver Windows drivers are compliant with Microsoft's Windows Certification Program
- Two months of free technical support.
- No run-time fees or royalties.

1.5. WinDriver Architecture

Figure 1.1. WinDriver Architecture



For hardware access, your application calls one of the WinDriver user-mode functions. The user-mode function calls the WinDriver kernel, which accesses the hardware for you through the native calls of the operating system.

1.6. What Platforms Does WinDriver Support?

WinDriver supports the following operating systems:

- Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP and Embedded Windows 8.1/8/7/XP — henceforth collectively: **Windows**
- Windows CE (a.k.a. Windows Embedded Compact) 4.x–7.x (including Windows Mobile) — henceforth collectively: **Windows CE**
- Linux

The same source code will run on all supported platforms — simply recompile it for the target platform. The source code is binary compatible across Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP; WinDriver executables can be ported among the binary-compatible platforms without recompilation.

Even if your code is meant only for one of the supported operating systems, using WinDriver will give you the flexibility to move your driver to another operating system in the future without needing to change your code.

1.7. Limitations of the Different Evaluation Versions

All the evaluation versions of the WinDriver USB Host toolkit are full featured. No functions are limited or crippled in any way. The evaluation version of WinDriver varies from the registered version in the following ways:

- Each time WinDriver is activated, an *Unregistered* message appears.
- When using DriverWizard, a dialogue box with a message stating that an evaluation version is being run appears on every interaction with the hardware.
- In the Linux and Windows CE versions, the driver will remain operational for 60 minutes, after which time it must be restarted.
- The Windows evaluation version expires 30 days from the date of installation.

For more details please refer to Appendix D.

1.8. How Do I Develop My Driver with WinDriver?

1.8.1. On Windows and Linux

1. Start DriverWizard and use it to diagnose your hardware — see details in Chapter 5.
2. Let DriverWizard generate skeletal code for your driver, or use one of the WinDriver samples as the basis for your driver application.
3. Modify the generated/sample code to suit your application's needs.
4. Run and debug your driver.

The code generated by DriverWizard is a diagnostics program that contains functions that perform data transfers on the device's pipes, send requests to the control pipe, change the active alternate setting, reset pipes, and more.

1.8.2. On Windows CE

1. Plug your hardware into a Windows host machine.
2. Diagnose your hardware using DriverWizard.
3. Let DriverWizard generate your driver's skeletal code.
4. Modify this code, using MS eMbedded Visual C++, to meet your specific needs. If you are using MS Platform Builder, activate it and insert the generated ***.pbp** into your project solution.
5. Test your driver on the target embedded Windows CE platorm.

1.9. What Does the WinDriver Toolkit Include?

- A printed version of this manual
- Two months of free technical support (Phone/Fax/Email)
- WinDriver modules
- Utilities
- Samples

1.9.1. WinDriver Modules

- WinDriver (**WinDriver/include**) — the general purpose hardware access toolkit. The main files here are
  - **windrvr.h**: Declarations and definitions of WinDriver's basic API.
  - **wdu_lib.h**: Declarations and definitions of the WinDriver USB (WDU) library, which provides convenient wrapper USB APIs.
  - **windrvr_int_thread.h**: Declarations of convenient wrapper functions to simplify interrupt handling.
  - **windrvr_events.h**: Declarations of APIs for handling and Plug-and-Play and power management events.
  - **utils.h**: Declarations of general utility functions.
  - **status_strings.h**: Declarations of API for converting WinDriver status codes to descriptive error strings.
- DriverWizard (**WinDriver/wizard/wdwizard**) — a graphical application that diagnoses your hardware and enables you to easily generate code for your driver (refer to Chapter 5 for details).
- Debug Monitor — a debugging tool that collects information about your driver as it runs. This tool is available both as a fully graphical application — **WinDriver/util/wddebug_gui** — and as a console-mode application — **WinDriver/util/wddebug**. The console-mode version also supports GUI execution on Windows CE platforms that don't have a command-line prompt. For details regarding the Debug Monitor, refer to Section 7.2.
- WinDriver distribution package (**WinDriver/redist**) — the files you include in the driver distribution to customers.
- This manual — the full WinDriver manual (this document), in different formats, can be found under the **WinDriver/docs** directory.

1.9.2. Utilities

- **usb_diag.exe** (**WinDriver/util/usb_diag.exe**) — enables the user to view the resources of connected USB devices and communicate with the devices — transfer data to/from the device, set the active alternate setting, reset pipes, etc.
  On Windows the program identifies all devices that have been registered to work with WinDriver using an INF file. On the other supported operating systems the program identifies all USB devices connected to the target platform.
- **pci_dump.exe** (**WinDriver/util/pci_dump.exe**) — used to obtain a dump of the PCI configuration registers of the installed PCI cards.

- **pci_scan.exe** (**WinDriver/util/pci_scan.exe**) — used to obtain a list of the PCI cards installed and the resources allocated for each card.
- **pcmcia_diag.exe** (**WinDriver/util/pcmcia_diag.exe**) — used for reading/writing PCMCIA attribute space, accessing PCMCIA I/O and memory ranges and handling PCMCIA interrupts.
- **pcmcia_scan.exe** (**WinDriver/util/pcmcia_scan.exe**) — used to obtain a list of the PCMCIA cards installed and the resources allocated for each card.

1.9.3. Samples

WinDriver includes a variety of samples that demonstrate how to use WinDriver's API to communicate with your device and perform various driver tasks.

- C samples: found under the **WinDriver/samples** directory.
  These samples also include the source code for the utilities listed above [1.9.2].
- .NET C# and Visual Basic .NET samples (Windows): found under the **WinDriver\csharp.net** and **WinDriver\vb.net** directories (respectively).

1.10. Can I Distribute the Driver Created with WinDriver?

Yes. WinDriver is purchased as a development toolkit, and any device driver created using WinDriver may be distributed, royalties free, in as many copies as you wish. See the license agreement at (**WinDriver/docs/wd_license.pdf**) for more details.

Chapter 2
Understanding Device Drivers

This chapter provides you with a general introduction to device drivers and takes you through the structural elements of a device driver.

Using WinDriver, you do not need to familiarize yourself with the internal workings of driver development. As explained in Chapter 1 of the manual, WinDriver enables you to communicate with your hardware and develop a driver for your device from the user mode, using only WinDriver's simple APIs, without any need for driver or kernel development knowledge.

2.1. Device Driver Overview

Device drivers are the software segments that provides an interface between the operating system and the specific hardware devices such as terminals, disks, tape drives, video cards and network media. The device driver brings the device into and out of service, sets hardware parameters in the device, transmits data from the kernel to the device, receives data from the device and passes it back to the kernel, and handles device errors.

A driver acts like a translator between the device and programs that use the device. Each device has its own set of specialized commands that only its driver knows. In contrast, most programs access devices by using generic commands. The driver, therefore, accepts generic commands from a program and then translates them into specialized commands for the device.

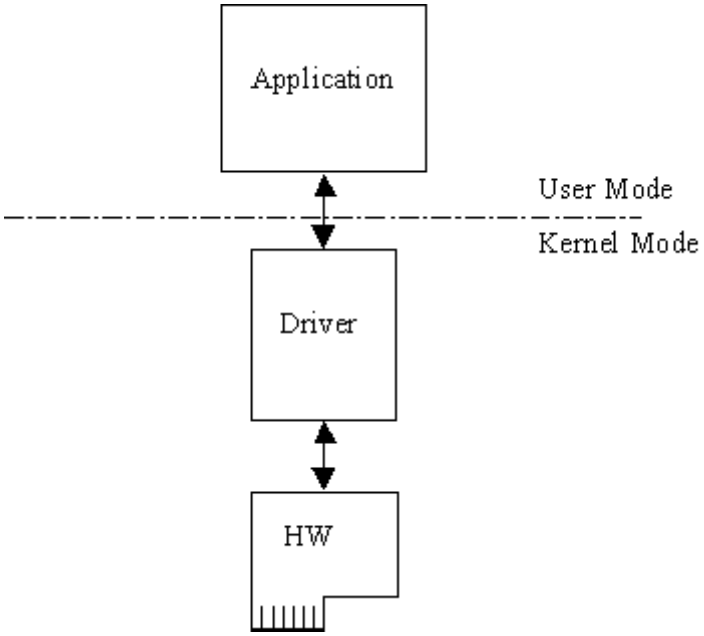
2.2. Classification of Drivers According to Functionality

There are numerous driver types, differing in their functionality. This subsection briefly describes three of the most common driver types.

2.2.1. Monolithic Drivers

Monolithic drivers are device drivers that embody all the functionality needed to support a hardware device. A monolithic driver is accessed by one or more user applications, and directly drives a hardware device. The driver communicates with the application through I/O control commands (IOCTLs) and drives the hardware using calls to the different WDK, ETK, DDI/DKI functions.

Figure 2.1. Monolithic Drivers



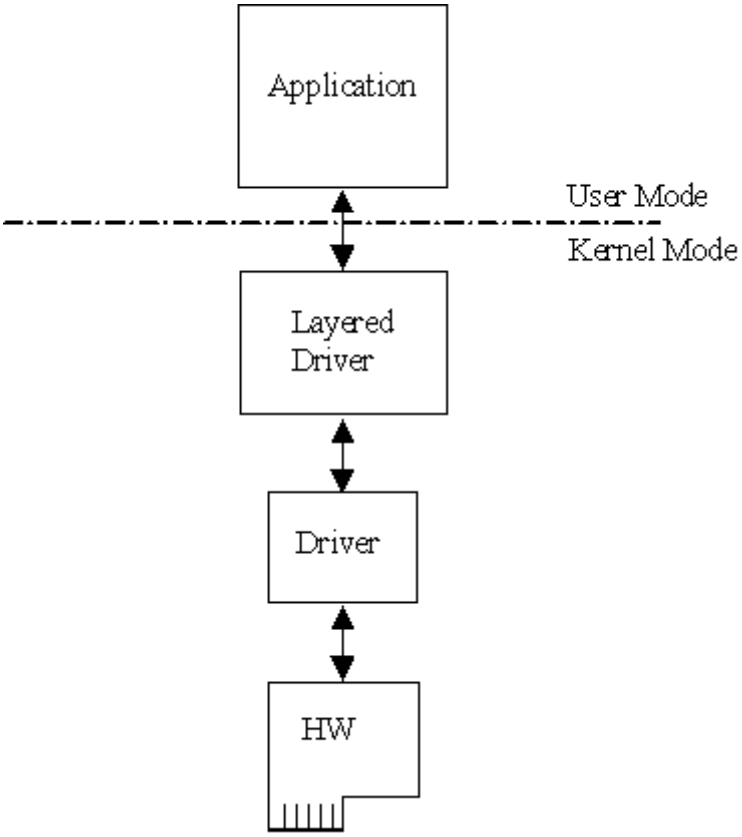
Monolithic drivers are supported in all operating systems including all Windows platforms and all Unix platforms.

2.2.2. Layered Drivers

Layered drivers are device drivers that are part of a stack of device drivers that together process an I/O request. An example of a layered driver is a driver that intercepts calls to the disk and encrypts/decrypts all data being transferred to/from the disk. In this example, a driver would be hooked on to the top of the existing driver and would only do the encryption/decryption.

Layered drivers are sometimes also known as filter drivers, and are supported in all operating systems including all Windows platforms and all Unix platforms.

Figure 2.2. Layered Drivers

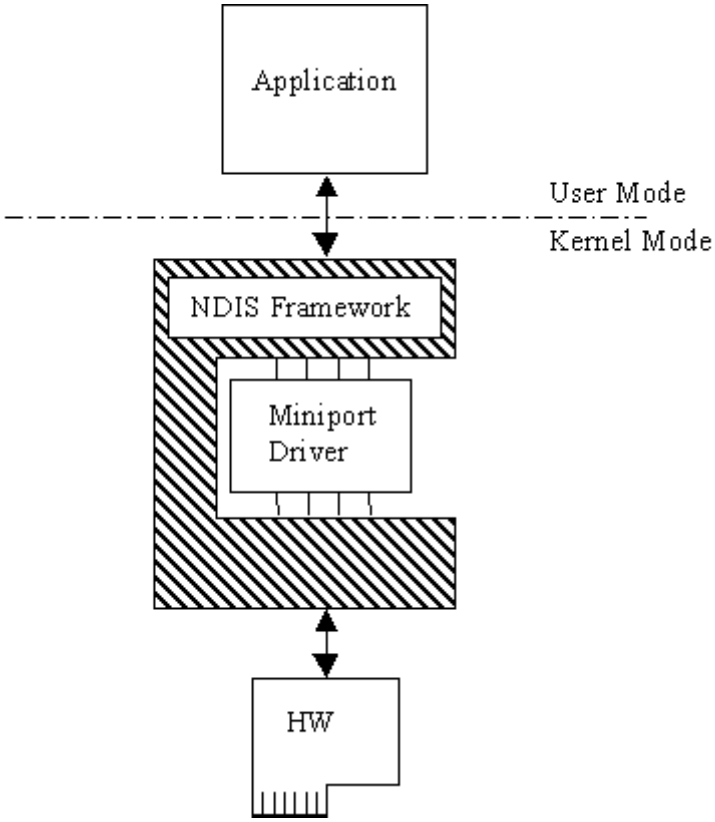


2.2.3. Miniport Drivers

A Miniport driver is an add-on to a class driver that supports miniport drivers. It is used so the miniport driver does not have to implement all of the functions required of a driver for that class. The class driver provides the basic class functionality for the miniport driver.
A class driver is a driver that supports a group of devices of common functionality, such as all HID devices or all network devices.

Miniport drivers are also called miniclass drivers or minidrivers, and are supported in the Windows XP and higher operating systems.

Figure 2.3. Miniport Drivers



The Windows XP and higher operating systems provide several driver classes (called ports) that handle the common functionality of their class. It is then up to the user to add only the functionality that has to do with the inner workings of the specific hardware. The NDIS miniport driver is one example of such a driver. The NDIS miniport framework is used to create network drivers that hook up to Windows's communication stacks, and are therefore accessible to common communication calls used by applications. The Windows kernel provides drivers for the various communication stacks and other code that is common to communication cards. Due to the NDIS framework, the network card developer does not have to write all of this code, only the code that is specific to the network card he is developing.

2.3. Classification of Drivers According to Operating Systems

2.3.1. WDM Drivers

Windows Driver Model (WDM) drivers are kernel-mode drivers within the Windows operating systems. WDM works by channeling some of the work of the device driver into portions of the code that are integrated into the operating system. These portions of code handle all of the low-level buffer management, including DMA and Plug-and-Play (Pnp) device enumeration. WDM drivers are PnP drivers that support power management protocols, and include monolithic drivers, layered drivers and miniport drivers.

2.3.2. Unix Device Drivers

In the classic Unix driver model, devices belong to one of three categories: character (char) devices, block devices and network devices. Drivers that implement these devices are correspondingly known as char drivers, block drivers or network drivers. Under Unix, drivers are code units linked into the kernel that run in privileged kernel mode. Generally, driver code runs on behalf of a user-mode application. Access to Unix drivers from user-mode applications is provided via the file system. In other words, devices appear to the applications as special device files that can be opened.

Unix device drivers are either layered or monolithic drivers. A monolithic driver can be perceived as a one-layer layered driver.

2.3.3. Linux Device Drivers

Linux device drivers are based on the classic Unix device driver model [2.3.2]. In addition, Linux introduces some new characteristics.

Under Linux, a block device can be accessed like a character device, as in Unix, but also has a block-oriented interface that is invisible to the user or application.

Traditionally, under Unix, device drivers are linked with the kernel, and the system is brought down and restarted after installing a new driver. Linux introduces the concept of a dynamically loadable driver called a module. Linux modules can be loaded or removed dynamically without requiring the system to be shut down. A Linux driver can be written so that it is statically linked or written in a modular form that allows it to be dynamically loaded. This makes Linux memory usage very efficient because modules can be written to probe for their own hardware and unload themselves if they cannot find the hardware they are looking for.

Like Unix device drivers, Linux device drivers are either layered or monolithic drivers.

2.4. The Entry Point of the Driver

Every device driver must have one main entry point, like the `main`() function in a C console application. This entry point is called `DriverEntry`() in Windows and `init_module`() in Linux. When the operating system loads the device driver, this driver entry procedure is called.

There is some global initialization that every driver needs to perform only once when it is loaded for the first time. This global initialization is the responsibility of the `DriverEntry`()/`init_module`() routine. The entry function also registers which driver callbacks will be called by the operating system. These driver callbacks are operating system requests for services from the driver. In Windows, these callbacks are called *dispatch routines*, and in Linux they are called *file operations*. Each registered callback is called by the operating system as a result of some criteria, such as disconnection of hardware, for example.

2.5. Associating the Hardware with the Driver

Operating systems differ in the ways they associate a device with a specific driver.

In Windows, the hardware-driver association is performed via an INF file, which registers the device to work with the driver. This association is performed before the `DriverEntry`() routine is called. The operating system recognizes the device, checks its database to identify which INF file is associated with the device, and according to the INF file, calls the driver's entry point.

In Linux, the hardware-driver association is defined in the driver's `init_module`() routine. This routine includes a callback that indicates which hardware the driver is designated to handle. The operating system calls the driver's entry point, based on the definition in the code.

2.6. Communicating with Drivers

Communication between a user-mode application and the driver that drives the hardware, is implemented differently for each operating system, using the custom OS Application Programming Interfaces (APIs).

On Windows, Windows CE, and Linux, the application can use the OS file-access API to open a handle to the driver (e.g., using the Windows `CreateFile`() function or using the Linux `open`() function), and then read and write from/to the device by passing the handle to the relevant OS file-access functions (e.g., the Windows `ReadFile`() and `WriteFile`() functions, or the Linux `read`() and `write`() functions).

The application sends requests to the driver via I/O control (IOCTL) calls, using the custom OS APIs provided for this purpose (e.g., the Windows `DeviceIoControl`() function, or the Linux `ioctl`() function).
The data passed between the driver and the application via the IOCTL calls is encapsulated using custom OS mechanisms. For example, on Windows the data is passed via an I/O Request Packet (IRP) structure, and is encapsulated by the I/O Manager.

Chapter 3
WinDriver USB Overview

This chapter explores the basic characteristics of the Universal Serial Bus (USB) and introduces WinDriver USB's features and architecture.

The references to the WinDriver USB toolkit in this chapter relate to the standard WinDriver USB toolkit for development of USB host drivers.

3.1. Introduction to USB

USB (Universal Serial Bus) is an industry standard extension to the PC architecture for attaching peripherals to the computer. It was originally developed in 1995 by leading PC and telecommunication industry companies, such as Intel, Compaq, Microsoft and NEC. USB was developed to meet several needs, among them the needs for an inexpensive and widespread connectivity solution for peripherals in general and for computer telephony integration in particular, an easy-to-use and flexible method of reconfiguring the PC, and a solution for adding a large number of external peripherals. The USB standard meets these needs.

The USB specification allows for the connection of a maximum of 127 peripheral devices (including hubs) to the system, either on the same port or on different ports.

USB also supports Plug-and-Play installation and hot swapping. The **USB 1.1** standard supports both isochronous and asynchronous data transfers and has dual speed data transfer: 1.5 Mb/s (megabits per second) for **low-speed** USB devices and 12 Mb/s for **full-speed** USB devices (much faster than the original serial port). Cables connecting the device to the PC can be up to five meters (16.4 feet) long. USB includes built-in power distribution for low power devices and can provide limited power (up to 500 mA of current) to devices attached on the bus.

The **USB 2.0** standard supports a signalling rate of 480 Mb/s, known as '**high-speed**', which is 40 times faster than the USB 1.1 full-speed transfer rate.
USB 2.0 is fully forward- and backward-compatible with USB 1.1 and uses existing cables and connectors.
USB 2.0 supports connections with PC peripherals that provide expanded functionality and require wider bandwidth. In addition, it can handle a larger number of peripherals simultaneously. USB 2.0 enhances the user's experience of many applications, including interactive gaming, broadband Internet access, desktop and Web publishing, Internet services and conferencing.

Because of its benefits (described also in Section 3.2 below), USB is currently enjoying broad market acceptance.

3.2. WinDriver USB Benefits

This section describes the main benefits of the USB standard and the WinDriver USB toolkit, which supports this standard:

- External connection, maximizing ease of use
- Self identifying peripherals supporting automatic mapping of function to driver and configuration
- Dynamically attachable and re-configurable peripherals
- Suitable for device bandwidths ranging from a few Kb/s to hundreds of Mb/s
- Supports isochronous as well as asynchronous transfer types over the same set of wires
- Supports simultaneous operation of many devices (multiple connections)
- Supports a data transfer rate of up to 480 Mb/s (high-speed) for USB 2.0 (for the operating systems that officially support this standard) and up to 12 Mb/s (full-speed) for USB 1.1
- Guaranteed bandwidth and low latencies; appropriate for telephony, audio, etc. (isochronous transfer may use almost the entire bus bandwidth)
- Flexibility: supports a wide range of packet sizes and a wide range of data transfer rates
- Robustness: built-in error handling mechanism and dynamic insertion and removal of devices with no delay observed by the user
- Synergy with PC industry; Uses commodity technologies
- Optimized for integration in peripheral and host hardware
- Low-cost implementation, therefore suitable for development of low-cost peripherals
- Low-cost cables and connectors
- Built-in power management and distribution
- Specific library support for custom USB HID devices

3.3. USB Components

The Universal Serial Bus (USB) consists of the following primary components:

- **USB Host**: The USB host platform is where the USB host controller is installed and where the client software/device driver runs. The *USB Host Controller* is the interface between the host and the USB peripherals. The host is responsible for detecting the insertion and removal of

USB devices, managing the control and data flow between the host and the devices, providing power to attached devices and more.

- **USB Hub**: A USB device that allows multiple USB devices to attach to a single USB port on a USB host. Hubs on the back plane of the hosts are called *root hubs*. Other hubs are called *external hubs*.

- **USB Function**: A USB device that can transmit or receive data or control information over the bus and that provides a function. A function is typically implemented as a separate peripheral device that plugs into a port on a hub using a cable. However, it is also possible to create a *compound device*, which is a physical package that implements multiple functions and an embedded hub with a single USB cable. A compound device appears to the host as a hub with one or more non-removable USB devices, which may have ports to support the connection of external devices.

3.4. Data Flow in USB Devices

During the operation of a USB device, the host can initiate a flow of data between the client software and the device.

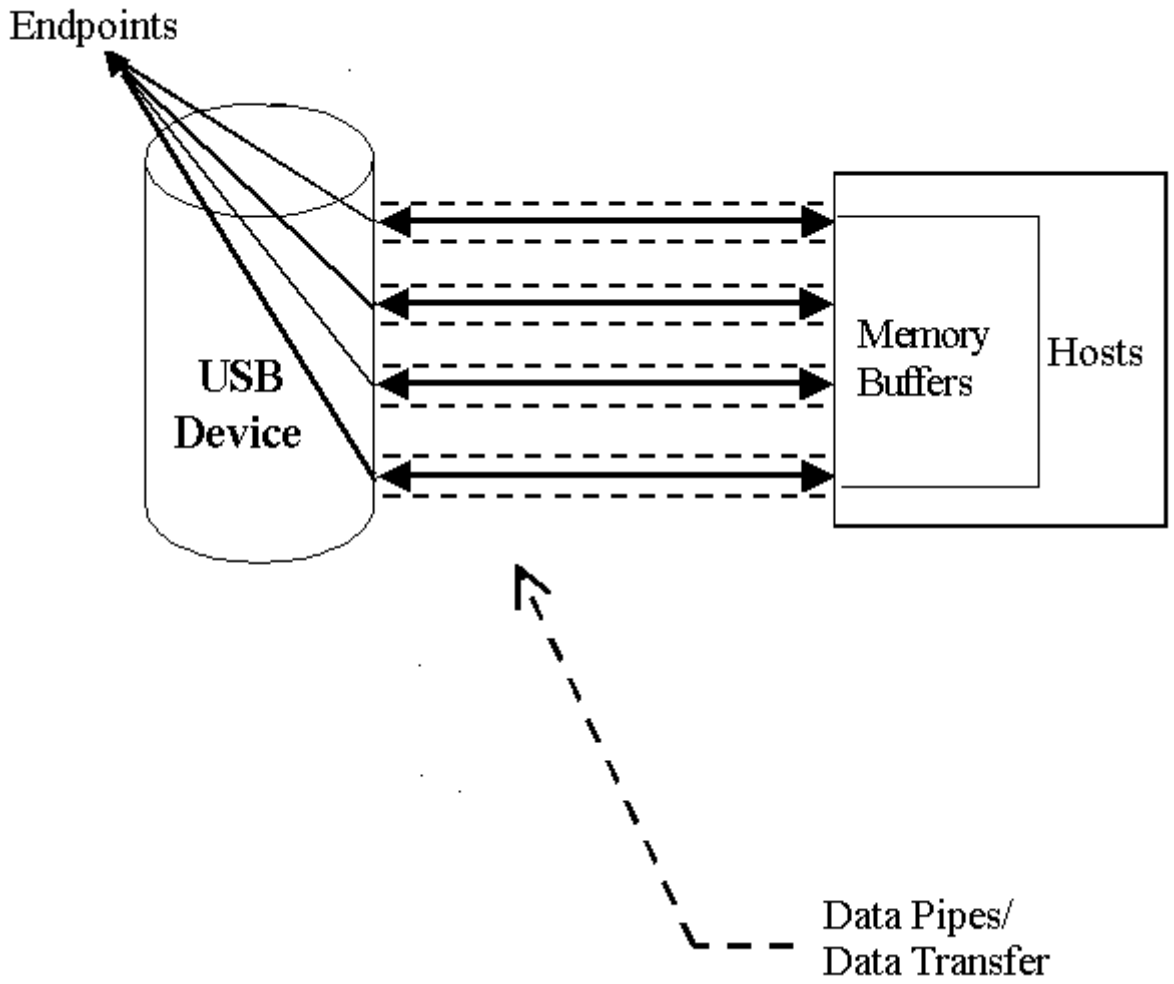
Data can be transferred between the host and only one device at a time (*peer to peer communication*). However, two hosts cannot communicate directly, nor can two USB devices (with the exception of On-The-Go (OTG) devices, where one device acts as the master (host) and the other as the slave.)

The data on the USB bus is transferred via pipes that run between software memory buffers on the host and endpoints on the device.

Data flow on the USB bus is half-duplex, i.e., data can be transmitted only in one direction at a given time.

An **endpoint** is a uniquely identifiable entity on a USB device, which is the source or terminus of the data that flows from or to the device. Each USB device, logical or physical, has a collection of independent endpoints. The three USB speeds (low, full and high) all support one bi-directional control endpoint (endpoint zero) and 15 unidirectional endpoints. Each unidirectional endpoint can be used for either inbound or outbound transfers, so theoretically there are 30 supported endpoints.
Each endpoint has the following attributes: bus access frequency, bandwidth requirement, endpoint number, error handling mechanism, maximum packet size that can be transmitted or received, transfer type and direction (into or out of the device).

Figure 3.1. USB Endpoints



A **pipe** is a logical component that represents an association between an endpoint on the USB device and software on the host. Data is moved to and from a device through a pipe. A pipe can be either a stream pipe or a message pipe, depending on the type of data transfer used in the pipe. *Stream pipes* handle interrupt, bulk and isochronous transfers, while *message pipes* support the control transfer type. The different USB transfer types are discussed below [3.6].

3.5. USB Data Exchange

The USB standard supports two kinds of data exchange between a host and a device: functional data exchange and control exchange.

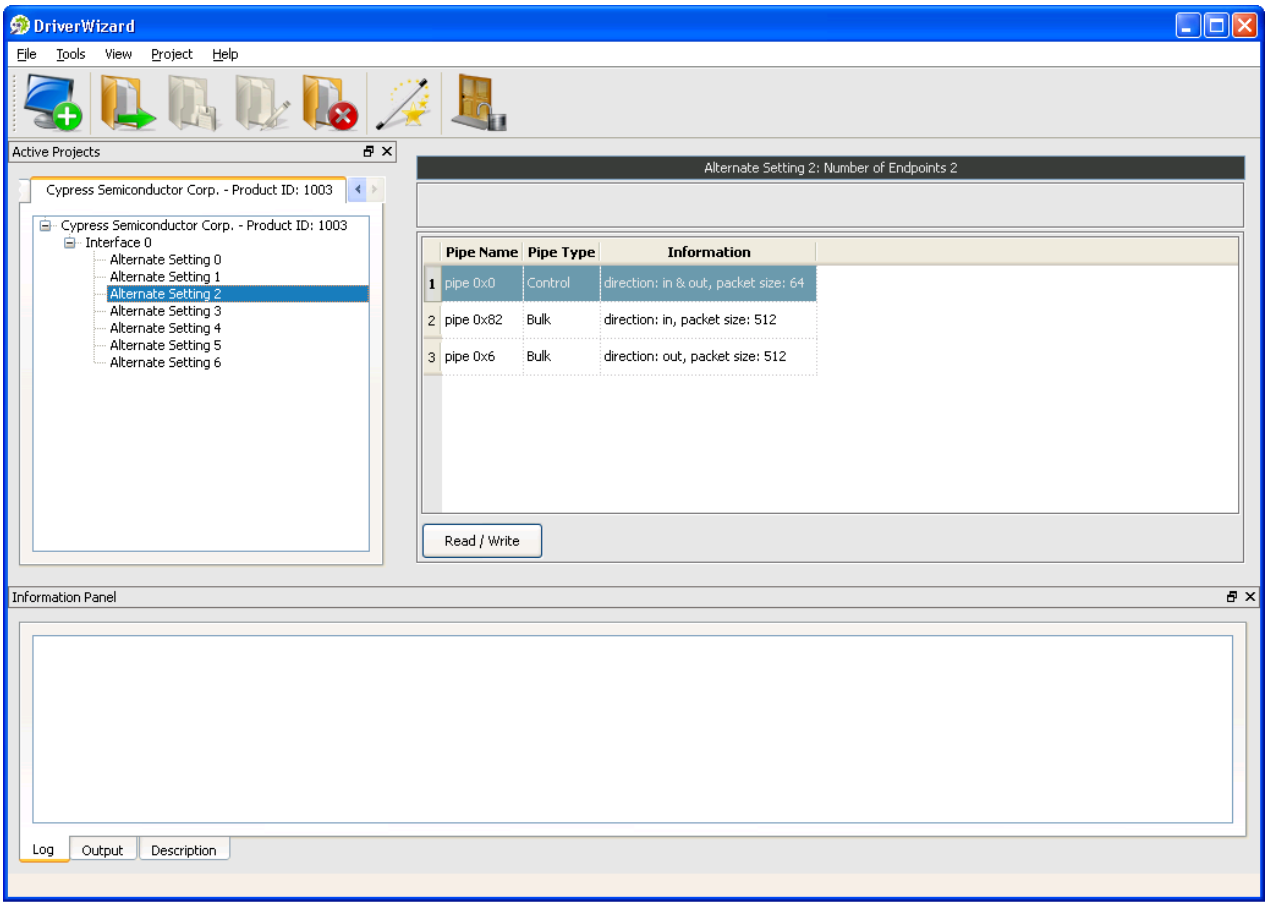
- **Functional Data Exchange** is used to move data to and from the device. There are three types of USB data transfers: Bulk, Interrupt and Isochronous .

- **Control Exchange** is used to determine device identification and configuration requirements and to configure a device, and can also be used for other device-specific purposes, including control of other pipes on the device.

Control exchange takes place via a control pipe — the default *pipe 0*, which always exists. The control transfer consists of a *setup stage* (in which a setup packet is sent from the host to the device), an optional *data stage* and a *status stage*.

Figure 3.2 below depicts a USB device with one bi-directional control pipe (endpoint) and two functional data transfer pipes (endpoints), as identified by WinDriver's DriverWizard utility (discussed in Chapter 5).

Figure 3.2. USB Pipes



More information on how to implement the control transfer by sending setup packets can be found in Section 8.2.

3.6. USB Data Transfer Types

The USB device (function) communicates with the host by transferring data through a pipe between a memory buffer on the host and an endpoint on the device. USB supports four different transfer types. A type is selected for a specific endpoint according to the requirements of the device and the software. The transfer type of a specific endpoint is determined in the endpoint descriptor.

The USB specification provides for the following data transfer types:

3.6.1. Control Transfer

Control Transfer is mainly intended to support configuration, command and status operations between the software on the host and the device.

This transfer type is used for low-, full- and high-speed devices.

Each USB device has at least one control pipe (default pipe), which provides access to the configuration, status and control information.

Control transfer is bursty, non-periodic communication.

The control pipe is bi-directional — i.e., data can flow in both directions.

Control transfer has a robust error detection, recovery and retransmission mechanism and retries are made without the involvement of the driver.

The maximum packet size for control endpoints can be only 8 bytes for low-speed devices; 8, 16, 32, or 64 bytes for full-speed devices; and only 64 bytes for high-speed devices.

For more in-depth information regarding USB control transfers and their implementation, refer to Section 8.2 of the manual.

3.6.2. Isochronous Transfer

Isochronous Transfer is most commonly used for time-dependent information, such as multimedia streams and telephony.

This transfer type can be used by full-speed and high-speed devices, but not by low-speed devices.

Isochronous transfer is periodic and continuous.

The isochronous pipe is unidirectional, i.e., a certain endpoint can either transmit or receive information. Bi-directional isochronous communication requires two isochronous pipes, one in each direction.

USB guarantees the isochronous transfer access to the USB bandwidth (i.e., it reserves the required amount of bytes of the USB frame) with bounded latency, and guarantees the data transfer rate through the pipe, unless there is less data transmitted.

Since timeliness is more important than correctness in this type of transfer, no retries are made in case of error in the data transfer. However, the data receiver can determine that an error occurred on the bus.

3.6.3. Interrupt Transfer

Interrupt Transfer is intended for devices that send and receive small amounts of data infrequently or in an asynchronous time frame.

This transfer type can be used for low-, full- and high-speed devices.

Interrupt transfer type guarantees a maximum service period and that delivery will be re-attempted in the next period if there is an error on the bus.

The interrupt pipe, like the isochronous pipe, is unidirectional and periodical.

The maximum packet size for interrupt endpoints can be 8 bytes or less for low-speed devices; 64 bytes or less for full-speed devices; and 1,024 bytes or less for high-speed devices.

3.6.4. Bulk Transfer

Bulk Transfer is typically used for devices that transfer large amounts of non-time sensitive data, and that can use any available bandwidth, such as printers and scanners.

This transfer type can be used by full-speed and high-speed devices, but not by low-speed devices.

Bulk transfer is non-periodic, large packet, bursty communication.

Bulk transfer allows access to the bus on an "as-available" basis, guarantees the data transfer but not the latency, and provides an error check mechanism with retries attempts. If part of the USB bandwidth is not being used for other transfers, the system will use it for bulk transfer.

Like the other stream pipes (isochronous and interrupt), the bulk pipe is also unidirectional, so bi-directional transfers require two endpoints.

The maximum packet size for bulk endpoints can be 8, 16, 32, or 64 bytes for full-speed devices, and 512 bytes for high-speed devices.

3.7. USB Configuration

Before the USB function (or functions, in a compound device) can be operated, the device must be configured. The host does the configuring by acquiring the configuration information from the USB device. USB devices report their attributes by descriptors. A **descriptor** is the defined structure and format in which the data is transferred. A complete description of the USB descriptors can be found in Chapter 9 of the USB Specification (see http://www.usb.org for the full specification).

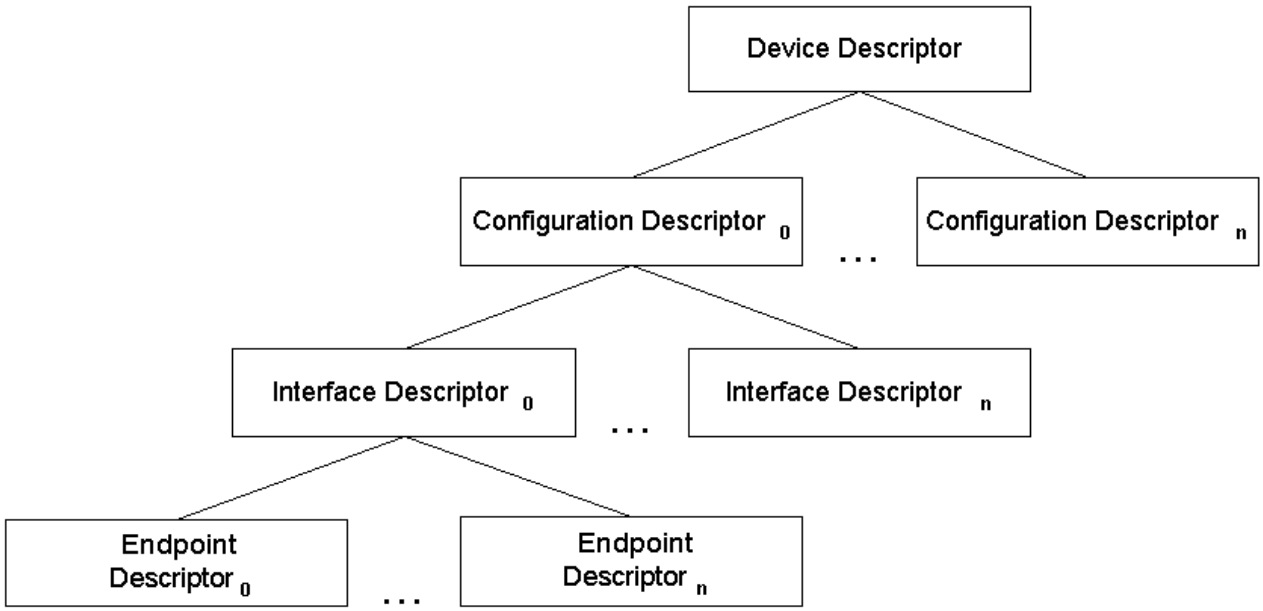
It is best to view the USB descriptors as a hierarchical structure with four levels:

- The *Device* level
- The *Configuration* level
- The *Interface* level (this level may include an optional sub-level called *Alternate Setting*)

- The *Endpoint* level

There is only one device descriptor for each USB device. Each device has one or more configurations, each configuration has one or more interfaces, and each interface has zero or more endpoints, as demonstrated in Figure 3.3 below.

Figure 3.3. Device Descriptors



- **Device Level**: The device descriptor includes general information about the USB device, i.e., global information for all of the device configurations. The device descriptor identifies, among other things, the device class (HID device, hub, locator device, etc.), subclass, protocol code, vendor ID, device ID and more. Each USB device has one device descriptor.

- **Configuration Level**: A USB device has one or more configuration descriptors. Each descriptor identifies the number of interfaces grouped in the configuration and the power attributes of the configuration (such as self-powered, remote wakeup, maximum power consumption and more). Only one configuration can be loaded at a given time. For example, an ISDN adapter might have two different configurations, one that presents it with a single interface of 128 Kb/s and a second that presents it with two interfaces of 64 Kb/s each.

- **Interface Level**: The interface is a related set of endpoints that present a specific functionality or feature of the device. Each interface may operate independently. The interface descriptor describes the number of the interface, the number of endpoints used by this interface and the interface-specific class, subclass and protocol values when the interface operates independently.

  In addition, an interface may have **alternate settings**. The alternate settings allow the endpoints or their characteristics to be varied after the device is configured.

- **Endpoint Level**: The lowest level is the endpoint descriptor, which provides the host with information regarding the endpoint's data transfer type and maximum packet size. For isochronous endpoints, the maximum packet size is used to reserve the required bus time for

the data transfer — i.e., the bandwidth. Other endpoint attributes are its bus access frequency, endpoint number, error handling mechanism and direction. The same endpoint can have different properties (and consequently different uses) in different alternate settings.

Seems complicated? Not at all! WinDriver automates the USB configuration process. The included DriverWizard utility [5] and USB diagnostics application scan the USB bus, detect all USB devices and their configurations, interfaces, alternate settings and endpoints, and enable you to pick the desired configuration before starting driver development.

WinDriver identifies the endpoint transfer type as determined in the endpoint descriptor. The driver created with WinDriver contains all configuration information acquired at this early stage.

3.8. WinDriver USB

WinDriver USB enables developers to quickly develop high-performance drivers for USB-based devices without having to learn the USB specifications and operating system internals, or use the operating system development kits. For example, Windows drivers can be developed without using the Windows Driver Kit (WDK) or learning the Windows Driver Model (WDM).

The driver code developed with WinDriver USB is binary compatible across the supported Windows platforms — Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP — and source code compatible across all supported operating systems — Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP, Embedded Windows 8.1/8/7/XP, Windows CE (a.k.a. Windows Embedded Compact) 4.x–7.x (including Windows Mobile), and Linux. For an up-to-date list of supported operating systems, visit Jungo's web site — http://www.jungo.com.

WinDriver USB is a generic tool kit that supports all USB devices from all vendors and with all types of configurations.

WinDriver USB encapsulates the USB specification and architecture, letting you focus on your application logic. WinDriver USB features the graphical DriverWizard utility [5], which enables you to easily detect your hardware, view its configuration information, and test it, before writing a single line of code: DriverWizard first lets you choose the desired configuration, interface and alternate setting combination, using a friendly graphical user interface. After detecting and configuring your USB device, you can proceed to test the communication with the device — perform data transfers on the pipes, send control requests, reset the pipes, etc. — in order to ensure that all your hardware resources function as expected.

After your hardware is diagnosed, you can use DriverWizard to automatically generate your device driver source code in C, Visual Basic .NET, or C#. WinDriver USB provides user-mode APIs, which you can call from within your application in order to implement the communication with your device. The WinDriver USB API includes USB-unique operations such as reset of a pipe or a device. The generated DriverWizard code implements a diagnostics application, which demonstrates how to use WinDriver's USB API to drive your specific device. In order to use the application you just need to compile and run it. You can jump-start your development cycle by using this application as your skeletal driver and then modifying the code, as needed, to implement the desired driver functionality for your specific device.

DriverWizard also automates the creation of an INF file that registers your device to work with WinDriver, which is an essential step in order to correctly identify and handle USB devices using WinDriver. For an explanation on why you need to create an INF file for your USB device, refer to Section 11.1.1 of the manual. For detailed information on creation of INF files with DriverWizard, refer to Section 5.2 (see specifically Step 3).

With WinDriver USB, all development is done in the user mode, using familiar development and debugging tools and your favorite compiler or development environment (such as MS Visual Studio, MS eMbedded Visual C++, MS Platform Builder C++, GCC, Windows GCC).

For more information regarding implementation of USB transfers with WinDriver, refer to Chapter 8 of the manual.

3.9. WinDriver USB Architecture

To access your hardware, your application calls the WinDriver kernel module using functions from the WinDriver USB API. The high-level functions utilize the low-level functions, which use IOCTLs to enable communication between the WinDriver kernel module and your user-mode application. The WinDriver kernel module accesses your USB device resources through the native operating system calls.

There are two layers responsible for abstracting the USB device to the USB device driver. The upper layer is the **USB Driver (USBD)** layer, which includes the USB Hub Driver and the USB Core Driver. The lower level is the **Host Controller Driver (HCD)** layer. The division of duties between the HCD and USBD layers is not defined and is operating system dependent. Both the HCD and USBD are software interfaces and components of the operating system, where the HCD layer represents a lower level of abstraction.

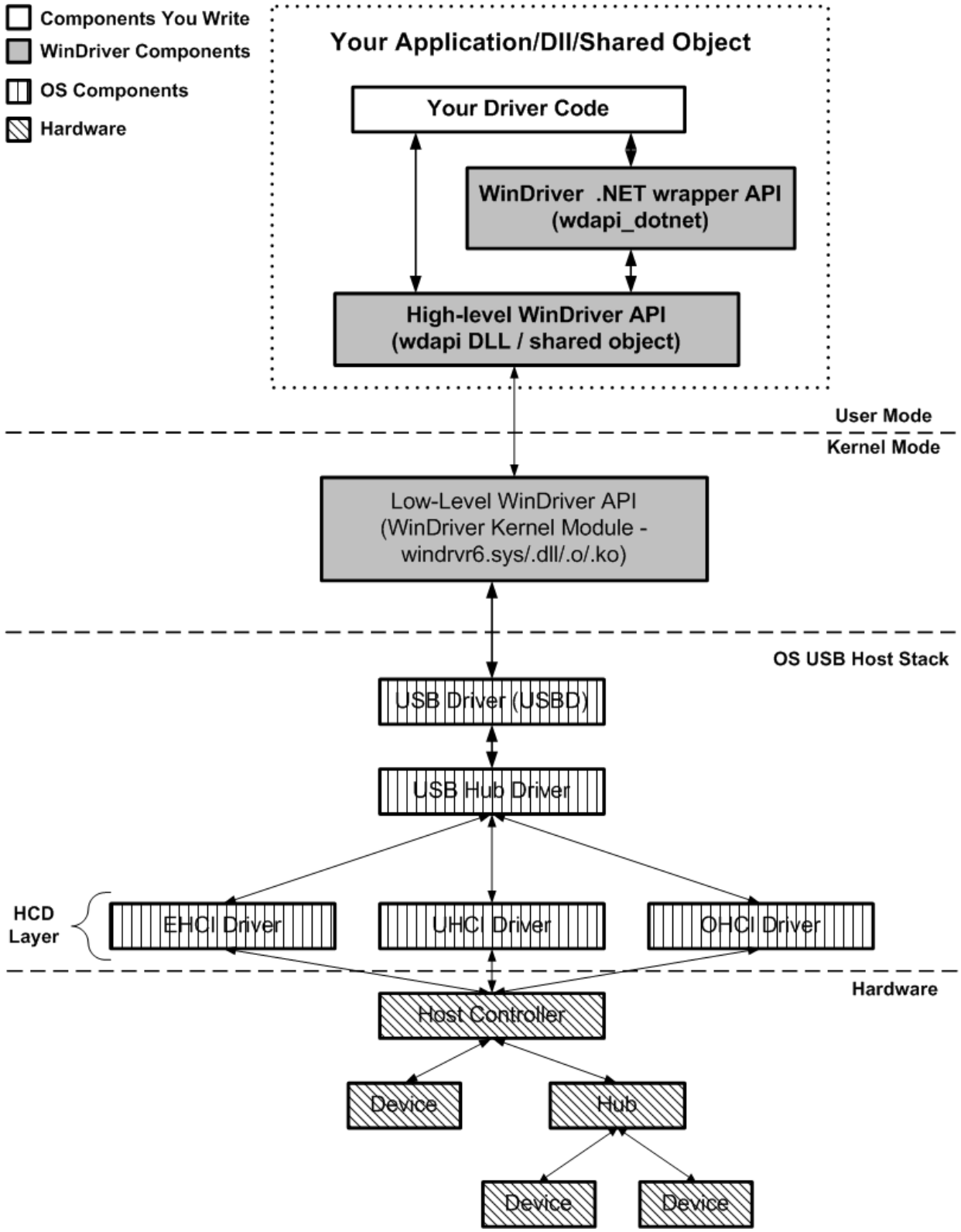
The **HCD** is the software layer that provides an abstraction of the host controller hardware, while the **USBD** provides an abstraction of the USB device and the data transfer between the host software and the function of the USB device.

The **USBD** communicates with its clients (the specific device driver, for example) through the USB Driver Interface (**USBDI**). At the lower level, the Core Driver and USB Hub Driver implement the hardware access and data transfer by communicating with the HCD using the Host Controller Driver Interface (**HCDI**).

The USB Hub Driver is responsible for identifying the addition and removal of devices from a particular hub. When the Hub Driver receives a signal that a device was attached or detached, it uses additional host software and the USB Core Driver to recognize and configure the device. The software implementing the configuration can include the hub driver, the device driver, and other software.

WinDriver USB abstracts the configuration procedure and hardware access described above for the developer. With WinDriver's USB API, developers can perform all the hardware-related operations without having to master the lower-level implementation for supporting these operations.

Figure 3.4. WinDriver USB Architecture



Chapter 4
Installing WinDriver

This chapter takes you through the process of installing WinDriver on your development platform, and shows you how to verify that your WinDriver is properly installed. The last section discusses the uninstall procedure. To find out how to install the driver you create on target platforms, refer to Chapter 10.

4.1. System Requirements

4.1.1. Windows System Requirements

- Any x86 32-bit or 64-bit (x64: AMD64 or Intel EM64T) processor
- Any compiler or development environment supporting C or .NET
- Windows XP requires at least SP2

4.1.2. Windows CE System Requirements

- An x86, MIPS, or ARM target platform running Windows CE (a.k.a. Windows Embedded Compact) 4.x–7.x (including Windows Mobile)
- Windows 8.1/Server 2012 R2/8/Server 2012/7/Server 2008 R2/Vista/Server 2008/Server 2003/XP host development platform
- IDE:
  - For **Windows CE 4.x–5.x** (including Windows Mobile):
    - Microsoft eMbedded Visual C++ or Microsoft Visual Studio 2005/2008, with a corresponding target SDK

    OR
    - Microsoft Platform Builder with a corresponding Board Support Package (BSP) for the target platform
  - For **Windows CE 6.x**: Microsoft Visual Studio 2005/2008 with a corresponding target SDK or with the Windows CE 6.0 plugin
  - For **Windows CE 7.x**: Microsoft Visual Studio 2008 with the Windows CE 7 plugin

4.1.3. Linux System Requirements

- Any of the following processor architectures, with a 2.6.x or higher Linux kernel:
  - 32-bit x86
  - 64-bit x86 AMD64 or Intel EM64T (**x86_64**)

Jungo strives to support new Linux kernel versions as close as possible to their release. To find out the latest supported kernel version, refer to the WinDriver release notes (found online at http://www.jungo.com/st/support/windriver/wdver/).

- A GCC compiler

The version of the GCC compiler should match the compiler version used for building the running Linux kernel.

- Any 32-bit or 64-bit development environment (depending on your target configuration) supporting C for user mode
- On your development PC: **glibc2.3.x**
- **libstdc++.so.5** — required for running GUI WinDriver applications (e.g., DriverWizard [5]; Debug Monitor [7.2])

4.2. WinDriver Installation Process

4.2.1. Windows WinDriver Installation Instructions

Driver installation on Windows requires administrator privileges.

1. Run the WinDriver installation — **WD1150.EXE** — and follow the installation instructions.
2. At the end of the installation, you may be prompted to reboot your computer.

- The WinDriver installation defines a **WD_BASEDIR** environment variable, which is set to point to the location of your WinDriver directory, as selected during the installation. This variable is used during the DriverWizard [5] code generation — it determines the default directory for saving your generated code and is used in the include paths of the generated project/make files.
- If the installation fails with an **ERROR_FILE_NOT_FOUND** error, inspect the Windows registry to see if the **RunOnce** key exists in **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion**. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the **RunOnce** key is missing, create it; then try installing the INF file again.

The following steps are for registered users only:

To register your copy of WinDriver with the license you received from Jungo, follow these steps:

3. Start DriverWizard: **Start | Programs | WinDriver | DriverWizard**.
4. Select the **Register WinDriver** option from the **File** menu, and insert the license string you received from Jungo.
5. Click the **Activate License** button.
6. To register source code that you developed during the evaluation period, refer to the documentation of `WDU_Init()` [B.4.1].

4.2.2. Windows CE WinDriver Installation Instructions

4.2.2.1. Installing WinDriver CE when Building New CE-Based Platforms

- The following instructions apply to platform developers who build Windows CE kernel images using Windows CE Platform Builder or using MS Visual Studio 2005/2008 with the appropriate Windows CE plugin. The instructions use the notation **'Windows CE IDE'** to refer to either of these platforms.
- We recommend that you read Microsoft's documentation and understand the Windows CE and device driver integration procedure before you perform the installation.

1. Modify the project registry file — **WinDriver\samples\wince_install\project_wd.reg** — to add an entry for your target device.
2. Compile your Windows CE platform (Sysgen stage).
3. Integrate the driver into your platform:

a. Run the Windows CE IDE and open your platform.

b. Select **Open Release Directory** from the **Build** menu.

c. Copy the WinDriver CE kernel file — **WinDriver\redist\<TARGET_CPU>\windrvr6.dll** — to the **%_FLATRELEASEDIR%** subdirectory on the target development platform (should be the current directory in the new command window).

d. Append the contents of **WinDriver\samples\wince_install\project_wd.reg** to the **%_FLATRELEASEDIR%\project.reg** registry file.

e. Copy the contents of the **WinDriver\samples\wince_install\project_wd.bib** file to the FILES section of the binary image builder file — **%_FLATRELEASEDIR%\project.bib**. Then uncomment the line that matches the target platform (see the "TODO" comments in the copied text).

This step is only necessary if you want the WinDriver CE kernel file (**windrvr6.dll**) to be a permanent part of the Windows CE image (**NK.BIN**), which is the case if you select to transfer the file to your target platform using a boot disk. If you prefer to have the file **windrvr6.dll** loaded on demand via the CESH/PPSH services, you do not need to perform this step until you build a permanent kernel.

4. Select **Make Run-Time Image** from the **Build** menu, and name the new image **NK.BIN**.

5. Download your new kernel to the target platform and initialize it either by selecting **Attach Device** from the **Target** menu, or by using a boot disk. For Windows CE 4.x, the menu is called **Download/Initialize** rather than **Attach Device**.

6. Restart your target CE platform. The WinDriver CE kernel will automatically load.

7. Compile and run the sample programs to make sure that WinDriver CE is loaded and is functioning correctly (see Section 4.4.2, which describes how to check your installation).

4.2.2.2. Installing WinDriver CE when Developing Applications for Windows CE Computers

Unless otherwise specified, 'Windows CE' references in this section include all supported Windows CE platforms, including Windows Mobile.

The following instructions apply to driver developers who do not build the Windows CE kernel, but only download their drivers, built using MS eMbedded Visual C++ or MS Visual Studio 2005/2008 to a ready-made Windows CE platform.

1. Run the WinDriver installation — **WD1150CE.EXE** — on your host Windows PC, and complete the installation.

2. Copy WinDriver's kernel module — **windrvr6.dll** — from the **WinDriver\redist\WINCE\<TARGET_CPU>** directory on the Windows host development PC to the **Windows** directory on your target Windows CE platform.

3. Add WinDriver to the list of device drivers Windows CE loads on boot:

   - Modify the registry according to the entries documented in the file **WinDriver\samples\wince_install\project_wd.reg**. This can be done using the Windows CE Pocket Registry Editor on the hand-held CE computer, or by using the Remote CE Registry Editor Tool supplied with MS eMbedded Visual C++ or MS Visual Studio 2005/2008. Note that in order to use the Remote CE Registry Editor tool you will need to have Windows CE Services installed on your Windows host platform.

   - On many versions of Windows CE, the operating system's security scheme prevents the loading of unsigned drivers at boot time, therefore the WinDriver kernel module has to be reloaded after boot. To load WinDriver on the target Windows CE platform every time the OS is started, copy the **WinDriver\redist\Windows_Mobile_5_ARMV4I\wdreg.exe** utility to the **Windows\StartUp** directory on the target PC.

4. Restart your target CE computer. The WinDriver CE kernel will automatically load. You will have to do a warm reset rather than just suspend/resume (use the reset or power button on your target CE computer).

5. Compile and run the sample programs to make sure that WinDriver CE is loaded and is functioning correctly (see Section 4.4, which describes how to check your installation).

4.2.2.3. Windows CE Installation Note

The WinDriver installation on the host Windows PC defines a **`WD_BASEDIR`** environment variable, which is set to point to the location of your WinDriver directory, as selected during the installation. This variable is used during the DriverWizard [5] code generation — it determines the default directory for saving your generated code, and is used in the include paths of the generated project/make files.

Note that if you install the WinDriver Windows toolkit on the same host PC, the installation will override the value of the `WD_BASEDIR` variable from the Windows CE installation.

4.2.3. Linux WinDriver Installation Instructions

4.2.3.1. Preparing the System for Installation

In Linux, kernel modules must be compiled with the same header files that the kernel itself was compiled with. Since WinDriver installs kernel modules, it must compile with the header files of the Linux kernel during the installation process.

Therefore, before you install WinDriver for Linux, verify that the Linux source code and the file **version.h** are installed on your machine:

Install the Linux kernel source code:

- If you have yet to install Linux, install it, including the kernel source code, by following the instructions for your Linux distribution.

- If Linux is already installed on your machine, check whether the Linux source code was installed. You can do this by looking for 'linux' in the **/usr/src** directory. If the source code is not installed, either install it, or reinstall Linux with the source code, by following the instructions for your Linux distribution.

Install version.h:

- The file **version.h** is created when you first compile the Linux kernel source code. Some distributions provide a compiled kernel without the file **version.h**. Look under **/usr/src/linux/include/linux** to see whether you have this file. If you do not, follow these steps:

  1. Become super user:
     ```
     $ su
     ```

  2. Change directory to the Linux source directory:
     ```
     # cd /usr/src/linux
     ```

  3. Type:
     ```
     # make xconfig
     ```

  4. Save the configuration by choosing **Save and Exit**.

  5. Type:
     ```
     # make dep
     ```

  6. Exit super user mode:
     ```
     # exit
     ```

To run GUI WinDriver applications (e.g., DriverWizard [5]; Debug Monitor [7.2]) you must also have version 5.0 of the **libstdc++** library — **libstdc++.so.5**. If you do not have this file, install it from the relevant RPM in your Linux distribution (e.g., **compat-libstdc++**).

Before proceeding with the installation, you must also make sure that you have a *linux* symbolic link. If you do not, create one by typing
```
/usr/src$ ln -s <target kernel>/linux
```
For example, for the Linux 2.4 kernel type
```
/usr/src$ ln -s linux-2.4/ linux
```

4.2.3.2. Installation

1. On your development Linux machine, change directory to your preferred installation directory, for example to your home directory:
   `$ cd ~`

   The path to the installation directory must not contain any spaces.

2. Extract the WinDriver distribution file — **WD1150LN.tgz** or **WD1150LNx86_64.tgz** —
   `$ tar xvzf <file location>/WD1150LN[x86_64].tgz`

   For example, to extract **WD1150LN.tgz** run this command:
   `$ tar xvzf ~/WD1150LN.tgz`

3. Change directory to your WinDriver **redist** directory (the tar automatically creates a **WinDriver** directory):
   `$ cd <WinDriver directory path>/redist`

4. Install WinDriver:

   a. **`<WinDriver directory>/redist$ ./configure`**

   - The configuration script creates a makefile based on the running kernel. You may select to use another installed kernel source, by executing the script with the **`--with-kernel-source=<path>`** option, where **`<path>`** is the full path to the kernel source directory — e.g., **/usr/src/linux**.
   - If the Linux kernel version is 2.6.26 or higher, the configuration script generates makefiles that use **kbuild** to compile the kernel modules. You can force the use of **kbuild** on earlier versions of Linux, by executing the configuration script with the **`--enable-kbuild`** flag.

   For a full list of the configuration script options, use the **`--help`** option:
   `./configure --help`

   b. **`<WinDriver directory>/redist$ make`**

   c. Become super user:
   `<WinDriver directory>/redist$ su`

   d. Install the driver:
   `<WinDriver directory>/redist# make install`

5. Create a symbolic link so that you can easily launch the DriverWizard GUI:
   `$ ln -s <path to WinDriver>/wizard/wdwizard /usr/bin/wdwizard`

6. Change the read and execute permissions on the file **wdwizard** so that ordinary users can access this program.

7. Change the user and group IDs and give read/write permissions to the device file **/dev/windrvr6**, depending on how you wish to allow users to access hardware through the device. Due to security reasons, by default the device file is created with permissions only for the root user. Change the permissions by modifying your **/etc/udev/permissions.d/50-udev.permissions** file. For example, add the following line to provide read and write permissions:
   ```
   windrvr6:root:root:0666
   ```

8. Define a new **WD_BASEDIR** environment variable and set it to point to the location of your WinDriver directory, as selected during the installation. This variable is used in the make and source files of the WinDriver samples and generated DriverWizard [5] code, and is also used to determine the default directory for saving your generated DriverWizard projects. If you do not define this variable you will be instructed to do so when attempting to build the sample/generated code using the WinDriver makefiles.

9. Exit super user mode:
   ```
   # exit
   ```

10. You can now start using WinDriver to access your hardware and generate your driver code!

Use the **WinDriver/util/wdreg** script to load the WinDriver kernel module [9.3].

The following steps are for registered users only:

To register your copy of WinDriver with the license you received from Jungo, follow these steps:

12. Start DriverWizard:
    ```
    $ <path to WinDriver>/wizard/wdwizard
    ```

13. Select the **Register WinDriver** option from the **File** menu, and insert the license string you received from Jungo.

14. Click the **Activate License** button.

15. To register source code that you developed during the evaluation period, refer to the documentation of `WDU_Init()` [B.4.1].

4.2.3.3. Restricting Hardware Access on Linux

Since **/dev/windrvr6** gives direct hardware access to user programs, it may compromise kernel stability on multi-user Linux systems. Please restrict access to DriverWizard and the device file **/dev/windrvr6** to trusted users.

For security reasons the WinDriver installation script does not automatically perform the steps of changing the permissions on **/dev/windrvr6** and the DriverWizard application (**wdwizard**).

4.3. Upgrading Your Installation

To upgrade to a new version of WinDriver on Windows, follow the steps outlined in Section 4.2.1, which illustrate the process of installing WinDriver for Windows. You can either choose to overwrite the existing installation or install to a separate directory.

After installation, start DriverWizard and enter the new license string, if you have received one. This completes the upgrade of WinDriver.

To upgrade your source code, pass the new license string as a parameter to `WDU_Init`() [B.4.1] (or to `WD_License`(), when using the old `WD_UsbXXX`() APIs).

The procedure for upgrading your installation on other operating systems is the same as the one described above. Please check the respective installation sections for installation details.

4.4. Checking Your Installation

4.4.1. Windows and Linux Installation Check

1. Start DriverWizard — **<path to WinDriver>/wizard/wdwizard**. On Windows you can also run DriverWizard from the **Start** menu: **Start | Programs | WinDriver | DriverWizard**.

2. If you are a registered user, make sure that your WinDriver license is registered (refer to Section 4.2, which explains how to install WinDriver and register your license).
   If you are an evaluation version user, you do not need to register a license.

4.4.2. Windows CE Installation Check

1. Copy the console-mode Debug Monitor utility — **WinDriver\util\wddebug\<TARGET_CPU>\wddebug.exe** — from the host Windows machine to a directory on your target Windows CE device.

2. Run the Debug Monitor with the **`status`** command on the target device:
   `wddebug.exe status`
   If the WinDriver installation was successful, the application will display information regarding the Debug Monitor version and current status, the running WinDriver kernel module, and general system information.

4.5. Uninstalling WinDriver

This section will help you to uninstall either the evaluation or registered version of WinDriver.

4.5.1. Windows WinDriver Uninstall Instructions

- You can select to use the graphical **wdreg_gui.exe** utility instead of **wdreg.exe**.
- **wdreg.exe** and **wdreg_gui.exe** are found in the **WinDriver\util** directory (see Chapter 9 for details regarding these utilities).

1. Close any open WinDriver applications, including DriverWizard, the Debug Monitor, and user-specific applications.

2. Uninstall all Plug-and-Play devices (USB/PCI/PCMCIA) that have been registered with WinDriver via an INF file:

   - Uninstall the device using the **wdreg** utility:
     `wdreg -inf <path to the INF file> uninstall`

   - Verify that no INF files that register your device(s) with WinDriver's kernel module (**windrvr6.sys**) are found in the **%windir%\inf** directory.

3. Uninstall WinDriver:

   - **On the development PC**, on which you installed the WinDriver toolkit:
     Run **Start | WinDriver | Uninstall** , **OR** run the **uninstall.exe** utility from the **WinDriver** installation directory.

     The uninstall will stop and unload the WinDriver kernel module (**windrvr6.sys**); delete the copy of the **windrvr6.inf** file from the **%windir%\inf** directory; delete WinDriver from Windows' **Start** menu; delete the **WinDriver** installation directory (except for files that you added to this directory); and delete the shortcut icons to the DriverWizard and Debug Monitor utilities from the Desktop.

- **On a target PC**, on which you installed the WinDriver kernel module (**windrvr6.sys**), but not the entire WinDriver toolkit:
  Use the **wdreg** utility to stop and unload the driver:
  `wdreg -inf <path to windrvr6.inf> uninstall`

  When running this command, **windrvr6.sys** should reside in the same directory as **windrvr6.inf**.

  (On the development PC, the relevant **wdreg** uninstall command is executed for you by the uninstall utility).

- If you attempt to uninstall WinDriver while there are open handles to the WinDriver service (**windrvr6.sys** or your renamed driver [11.2], or there are connected and enabled Plug-and-Play devices that are registered to work with this service, **wdreg** will fail to uninstall the driver. This ensures that you do not uninstall the driver while it is being used.

- You can check if the WinDriver kernel module is loaded by running the Debug Monitor utility (**WinDriver\util\wddebug_gui.exe**) [7.2]. When the driver is loaded, the Debug Monitor log displays driver and OS information; otherwise, it displays a relevant error message. On the development PC, the uninstall command will delete the Debug Monitor executables; to use this utility after the uninstallation, create a copy of **wddebug_gui.exe** before performing the uninstall procedure.

4. If **windrvr6.sys** was successfully unloaded, erase the following files (if they exist):

   - **%windir%\system32\drivers\windrvr6.sys**
   - **%windir%\inf\windrvr6.inf**
   - **%windir%\system32\wdapi1150.dll**
   - **%windir%\sysWOW64\wdapi1150.dll** (Windows x64)

5. Reboot the computer.

4.5.2. Linux WinDriver Uninstall Instructions

The following commands must be executed with root privileges.

1. Verify that the WinDriver driver modules are not being used by another program:
   - View the list of modules and the programs using each of them:
     `# /sbin/lsmod`
   - Identify any applications and modules that are using the WinDriver driver modules. (By default, WinDriver module names begin with **windrvr6**).
   - Close any applications that are using the WinDriver driver modules.
2. Run the following command to unload the WinDriver driver modules:
   `# /sbin/modprobe -r windrvr6`
3. Remove the file **.windriver.rc** from the **/etc** directory:
   `# rm -f /etc/.windriver.rc`
4. Remove the file **.windriver.rc** from **$HOME**:
   `# rm -f $HOME/.windriver.rc`
5. If you created a symbolic link to DriverWizard, remove the link using the command
   `# rm -f /usr/bin/wdwizard`
6. Remove the WinDriver installation directory using the command
   `# rm -rf <path to the WinDriver directory>`
   (for example: `# rm -rf ~/WinDriver`).
7. Remove the WinDriver shared object file, if it exists:
   /usr/lib/libwdapi1150.so (32-bit x86) /
   /usr/lib64/libwdapi1150.so (64-bit x86).

Chapter 5
Using DriverWizard

This chapter describes the WinDriver DriverWizard utility and its hardware diagnostics and driver code generation capabilities.

5.1. An Overview

DriverWizard (included in the WinDriver toolkit) is a graphical user interface (GUI) tool that is targeted at two major phases in the hardware and driver development:

- **Hardware diagnostics** — DriverWizard enables you to write and read hardware resources before writing a single line of code. After the hardware has been built, attach your device to a USB port on your machine, view its configuration and pipes information, and verify the hardware's functionality by transferring data on the pipes, sending standard requests on the control pipe, and resetting the pipes.

- **Code generation** — Once you have verified that the device is operating to your satisfaction, use DriverWizard generate skeletal driver source code with functions to view and access your hardware's resources.

On Windows, DriverWizard can also be used to generate an INF file [11.1] for your hardware.

The code generated by DriverWizard is composed of the following elements:

- **Library functions** for accessing each element of your device's resources (memory ranges, I/O ranges, registers and interrupts).

- **A 32-bit diagnostics program** in console mode with which you can diagnose your device. This application utilizes the special library functions described above. Use this diagnostics program as your skeletal device driver.

- **A project solution** that you can use to automatically load all of the project information and files into your development environment.
  For Linux, DriverWizard generates the required makefile.

5.2. DriverWizard Walkthrough

To use DriverWizard, follow these steps:

1. **Attach your hardware to the computer:**
   Attach your device to a USB port on your computer.

2. **Run DriverWizard and select your device:**

   a. Start DriverWizard — **<path to WinDriver>/wizard/wdwizard**. On Windows you can also run DriverWizard from the **Start** menu: **Start | Programs | WinDriver | DriverWizard**.

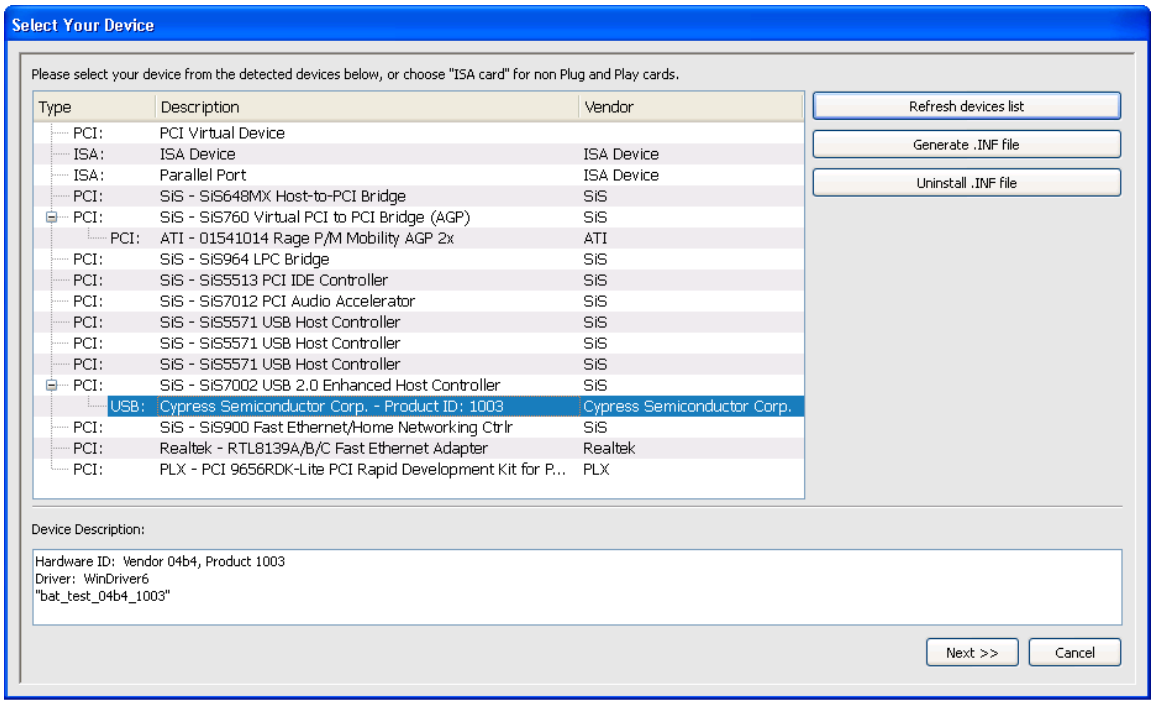
   On Windows Vista and higher you must run DriverWizard as administrator.

   b. Click **New host driver project** to start a new project, or **Open an existing project** to open a saved session.

Figure 5.1. Create or Open a WinDriver Project

c. Select your device from the list of devices detected by DriverWizard.

Figure 5.2. Select Your Device


Select Your Device
Please select your device from the detected devices below, or choose "ISA card" for non Plug and Play cards.
Type
Description
Vendor
PCI: PCI Virtual Device
ISA: ISA Device ISA Device
ISA: Parallel Port ISA Device
PCI: SiS - SiS648MX Host-to-PCI Bridge SiS
PCI: SiS - SiS760 Virtual PCI to PCI Bridge (AGP) SiS
PCI: ATI - 01541014 Rage P/M Mobility AGP 2x ATI
PCI: SiS - SiS964 LPC Bridge SiS
PCI: SiS - SiS5513 PCI IDE Controller SiS
PCI: SiS - SiS7012 PCI Audio Accelerator SiS
PCI: SiS - SiS5571 USB Host Controller SiS
PCI: SiS - SiS5571 USB Host Controller SiS
PCI: SiS - SiS5571 USB Host Controller SiS
PCI: SiS - SiS7002 USB 2.0 Enhanced Host Controller SiS
USB: Cypress Semiconductor Corp. - Product ID: 1003 Cypress Semiconductor Corp.
PCI: SiS - SiS900 Fast Ethernet/Home Networking Ctrlr SiS
PCI: Realtek - RTL8139A/B/C Fast Ethernet Adapter Realtek
PCI: PLX - PCI 9656RDK-Lite PCI Rapid Development Kit for P... PLX
Refresh devices list
Generate .INF file
Uninstall .INF file
Device Description:
Hardware ID: Vendor 04b4, Product 1003
Driver: WinDriver6
"bat_test_04b4_1003"
Next >>
Cancel

3. **Generate and install an INF file for your device [Windows]**:
   On the supported **Windows** operating systems, the driver for Plug-and-Play devices (such as USB) is installed by installing an INF file for the device. DriverWizard enables you to generate an INF file that registers your device to work with WinDriver (i.e., with the **windrvr6.sys** driver). The INF file generated by DriverWizard should later be distributed to your Windows customers, and installed on their PCs.
   The INF file that you generate in this step is also designed to enable DriverWizard to diagnose Plug-and-Play devices on Windows. Additional information concerning the need for an INF file is provided in Section 11.1.1.

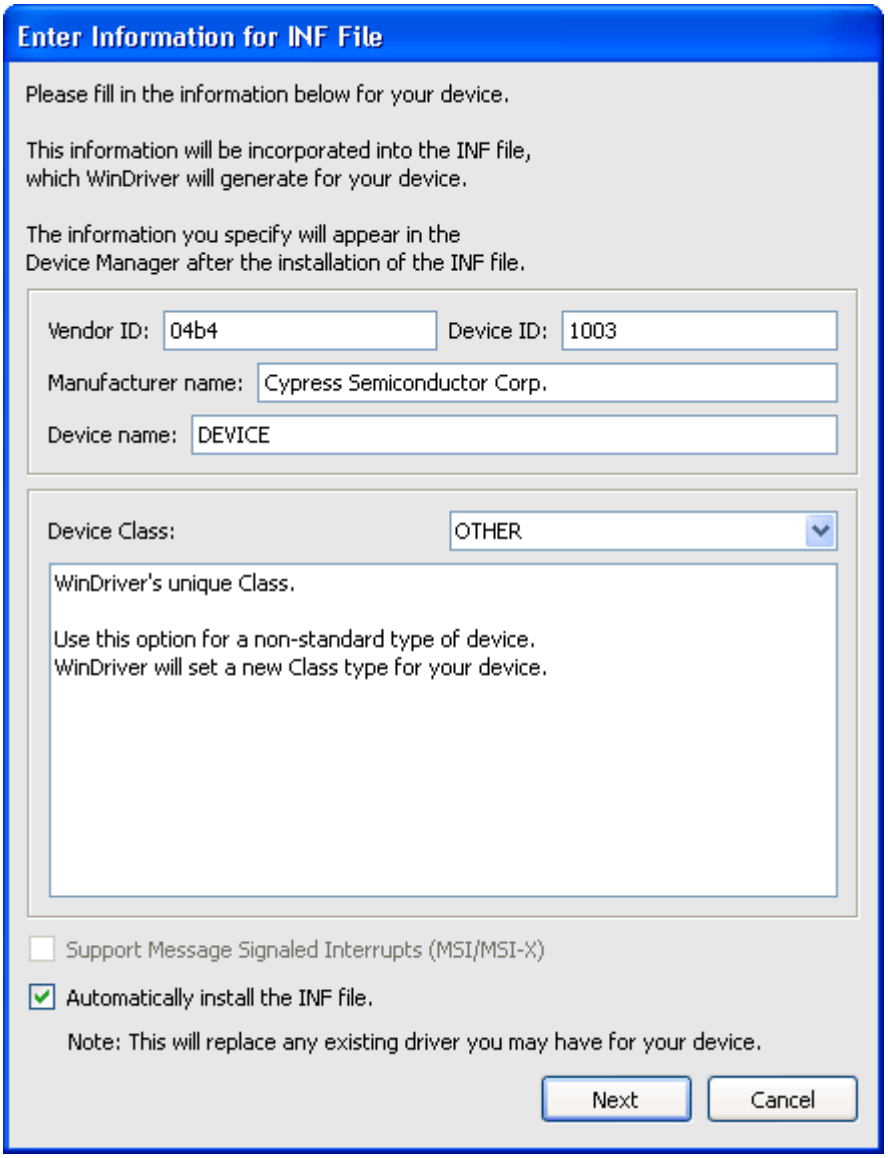
   If you don't need to generate and install an INF file, skip this step.

   To generate and install the INF file with DriverWizard, do the following:

   a. In the **Select Your Device** screen (see Step 2), click the **Generate .INF file** button or click **Next**.

   b. DriverWizard will display information detected for your device — Vendor ID, Product ID, Device Class, manufacturer name and device name — and allow you to modify this information.

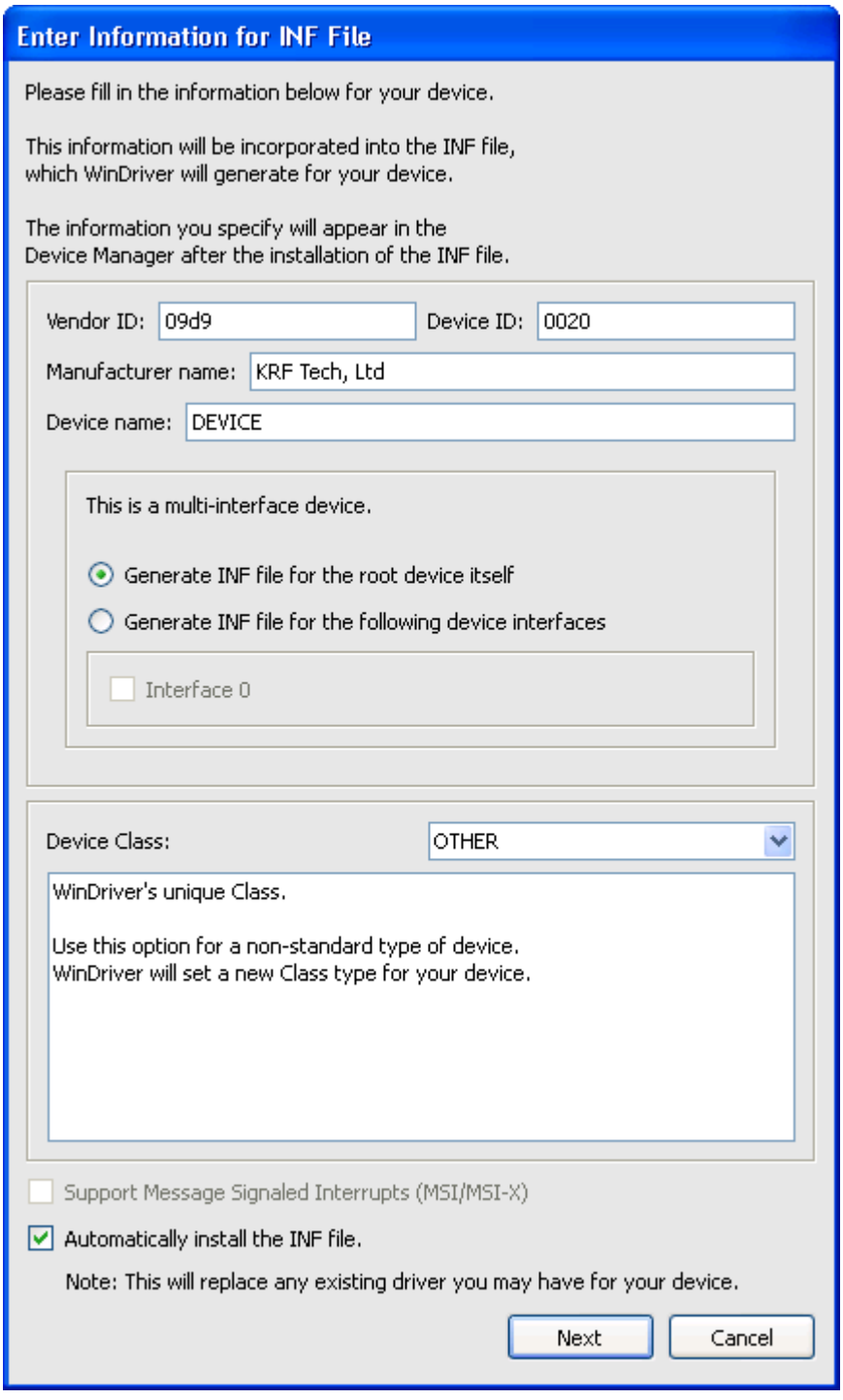
Figure 5.3. DriverWizard INF File Information


Enter Information for INF File
Please fill in the information below for your device.
This information will be incorporated into the INF file,
which WinDriver will generate for your device.
The information you specify will appear in the
Device Manager after the installation of the INF file.
Vendor ID:
04b4
Device ID:
1003
Manufacturer name:
Cypress Semiconductor Corp.
Device name:
DEVICE
Device Class:
OTHER
WinDriver's unique Class.
Use this option for a non-standard type of device.
WinDriver will set a new Class type for your device.
Support Message Signaled Interrupts (MSI/MSI-X)
Automatically install the INF file.
Note: This will replace any existing driver you may have for your device.
Next
Cancel

c. For multiple-interface USB devices, you can select to generate an INF file either for the composite device or for a specific interface.

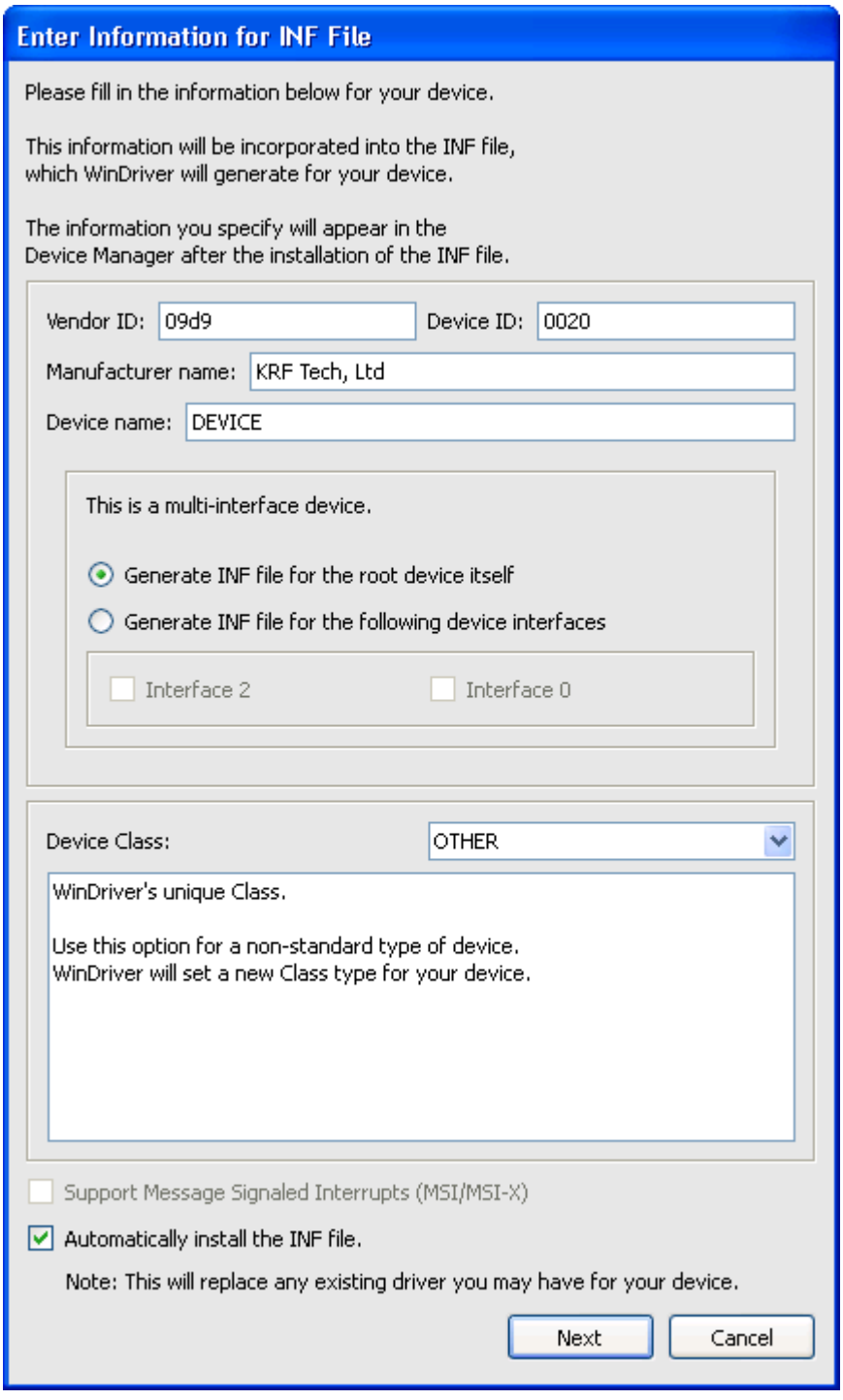
   - When selecting to generate an INF file for a specific interface of a multi-interface USB device the INF information dialogue will indicate for which interface the INF file is generated.

Figure 5.4. DriverWizard Multi-Interface INF File Information — Specific Interface


Enter Information for INF File
Please fill in the information below for your device.
This information will be incorporated into the INF file, which WinDriver will generate for your device.
The information you specify will appear in the Device Manager after the installation of the INF file.
Vendor ID: 09d9
Device ID: 0020
Manufacturer name: KRF Tech, Ltd
Device name: DEVICE
This is a multi-interface device.
Generate INF file for the root device itself
Generate INF file for the following device interfaces
Interface 0
Device Class: OTHER
WinDriver's unique Class.
Use this option for a non-standard type of device.
WinDriver will set a new Class type for your device.
Support Message Signaled Interrupts (MSI/MSI-X)
Automatically install the INF file.
Note: This will replace any existing driver you may have for your device.
Next
Cancel

- When selecting to generate an INF file for a composite device of a multi-interface USB device, the INF information dialogue provides you with the option to either generate an INF file for the root device itself, or generate an INF file for specific interfaces, which you can select from the dialogue.
  Selecting to generate an INF file for the root device will enable you to handle multiple active interfaces simultaneously.

Figure 5.5. DriverWizard Multi-Interface INF File Information — Composite Device


Enter Information for INF File
Please fill in the information below for your device.
This information will be incorporated into the INF file, which WinDriver will generate for your device.
The information you specify will appear in the Device Manager after the installation of the INF file.
Vendor ID: 09d9
Device ID: 0020
Manufacturer name: KRF Tech, Ltd
Device name: DEVICE
This is a multi-interface device.
Generate INF file for the root device itself
Generate INF file for the following device interfaces
Interface 2
Interface 0
Device Class: OTHER
WinDriver's unique Class.
Use this option for a non-standard type of device.
WinDriver will set a new Class type for your device.
Support Message Signaled Interrupts (MSI/MSI-X)
Automatically install the INF file.
Note: This will replace any existing driver you may have for your device.
Next
Cancel

d. When you are done, click **Next** and choose the directory in which you wish to store the generated INF file. DriverWizard will then automatically generate the INF file for you.

   You can choose to automatically install the INF file by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation dialogue.

If the automatic INF file installation fails, DriverWizard will notify you and provide manual installation instructions (refer also the manual INF file installation instructions in Section 11.1).

e. When the INF file installation completes, select and open your device from the list in the **Select Your Device** screen.

4. **Uninstall the INF file of your device [Windows]:**
On Windows, you can use DriverWizard to uninstall a previously installed device INF file. This will unregister the device from its current driver and delete the copy of the INF file in the Windows INF directory.

In order for WinDriver to correctly identify the resouces of a USB device and communicate with it — including for the purpose of the DriverWizard device diagnostics outlined in the next step — the deivce must be registered to work with WinDriver via an INF file (see Step 3).

If you do not wish to uninstall an INF file, skip this step.

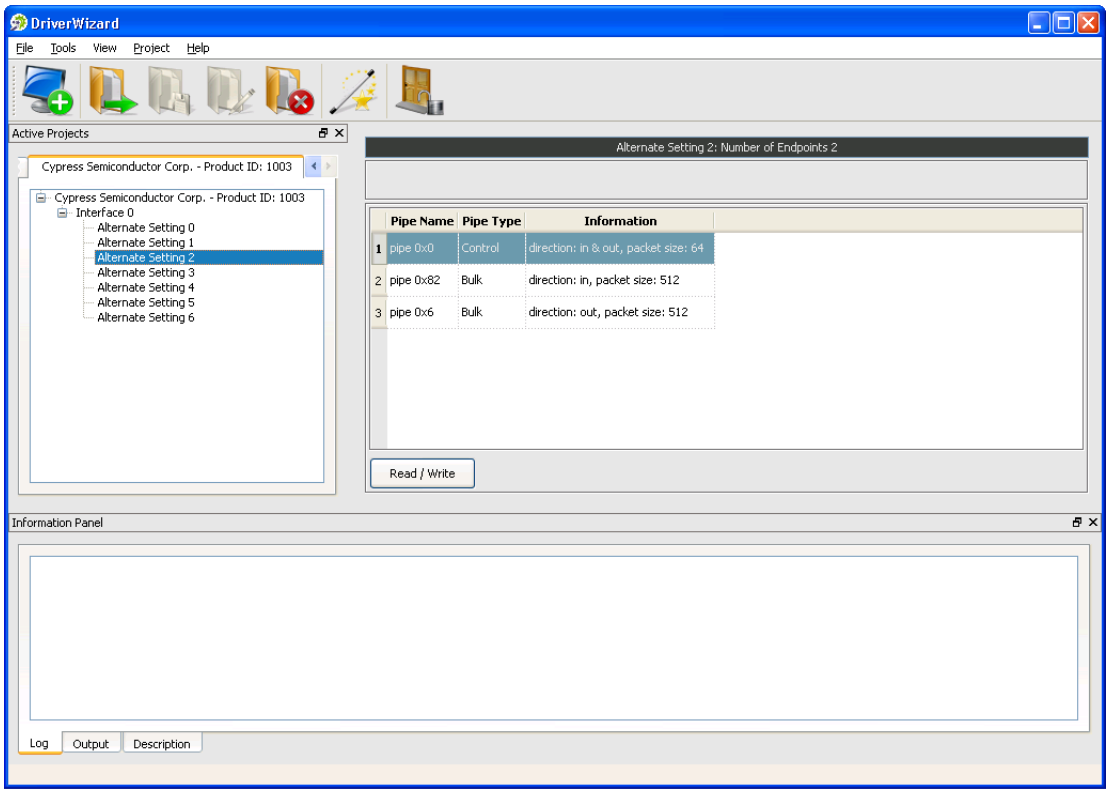
To uninstall the INF file, do the following:

a. In the **Select Your Device** screen (see Step 2), click the **Uninstall .INF file** button.

b. Select the INF file to be removed.

5. **Select the desired alternate setting:**

Figure 5.6. Select Device Interface



DriverWizard detects all the device's supported alternate settings and displays them, as demonstrated in Figure 5.6 below.
Select the desired **alternate setting** from the displayed list.

DriverWizard will display the pipes information for the selected alternate setting.

For USB devices with only one alternate setting configured, DriverWizard automatically selects the detected alternate setting and therefore the **Select Device Interface** dialogue will not be displayed.

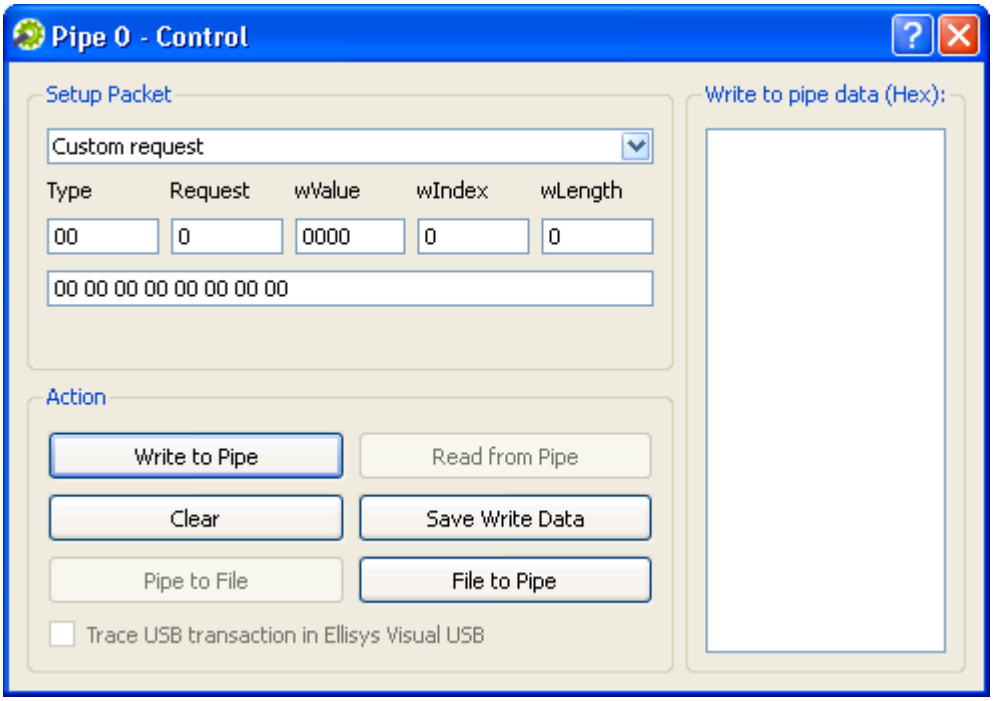
6. **Diagnose your device:**
   Before writing your device driver, it is important to make sure your hardware is working as expected. Use DriverWizard to diagnose your hardware. All of your activity will be logged in the DriverWizard log so that you may later analyze your tests:

   a. Test your USB device's pipes: DriverWizard shows the pipes detected for the selected alternate setting. To perform USB data transfers on the pipes, follow these steps:

      i. Select the desired pipe.

      ii. For a control pipe (a bidirectional pipe), click **Read / Write**. A new dialogue will appear, allowing you to select a standard USB request or define a custom request, as demonstrated in Figure 5.7.

Figure 5.7. USB Control Transfers



When you select one of the available standard USB requests, the setup packet information for the selected request is automatically filled and the request description is displayed in the **Request Description** box.

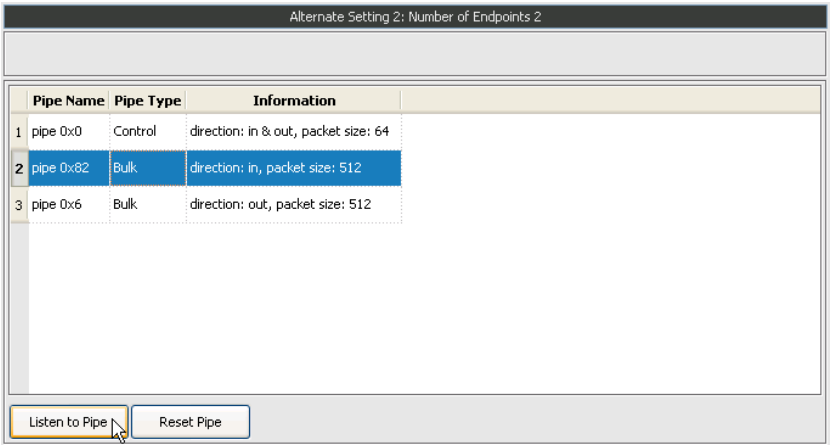
For a custom request, you are required to enter the setup packet information and write data (if exists) yourself. The size of the setup packet should be eight bytes and it should be defined using little endian byte ordering. The setup packet information should conform to the USB specification parameters (`bmRequestType`, `bRequest`, `wValue`, `wIndex`, `wLength`).

More detailed information on the standard USB requests, on how to implement the control transfer and how to send setup packets can be found in Section 8.2.

iii. For an input pipe (moves data from device to host) click **Listen to Pipe**. To successfully accomplish this operation with devices other than HID, you need to first verify that the device sends data to the host. If no data is sent after listening for a short period of time, DriverWizard will notify you that the **Transfer Failed**.

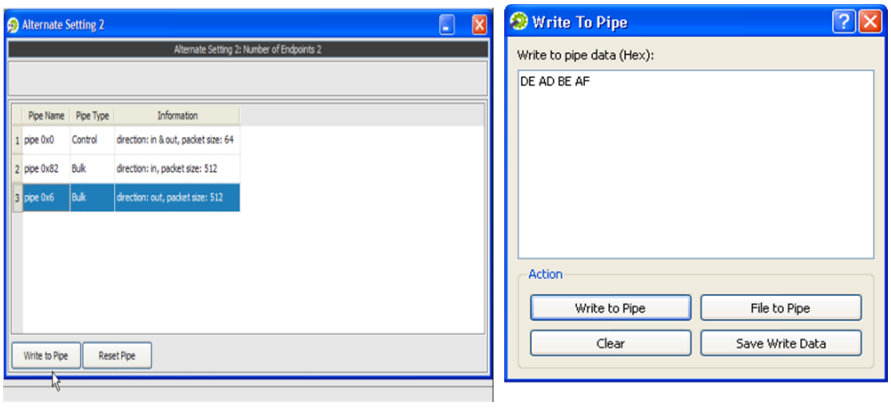
To stop reading, click **Stop Listen to Pipe**.

Figure 5.8. Listen to Pipe


Alternate Setting 2: Number of Endpoints 2
Pipe Name
Pipe Type
Information
1 pipe 0x0 Control direction: in & out, packet size: 64
2 pipe 0x82 Bulk direction: in, packet size: 512
3 pipe 0x6 Bulk direction: out, packet size: 512
Listen to Pipe
Reset Pipe

iv. For an output pipe (moves data from host to device), click **Write to Pipe**. A new dialogue box will appear asking you to enter the data to write. The DriverWizard log will contain the result of the operation.

Figure 5.9. Write to Pipe



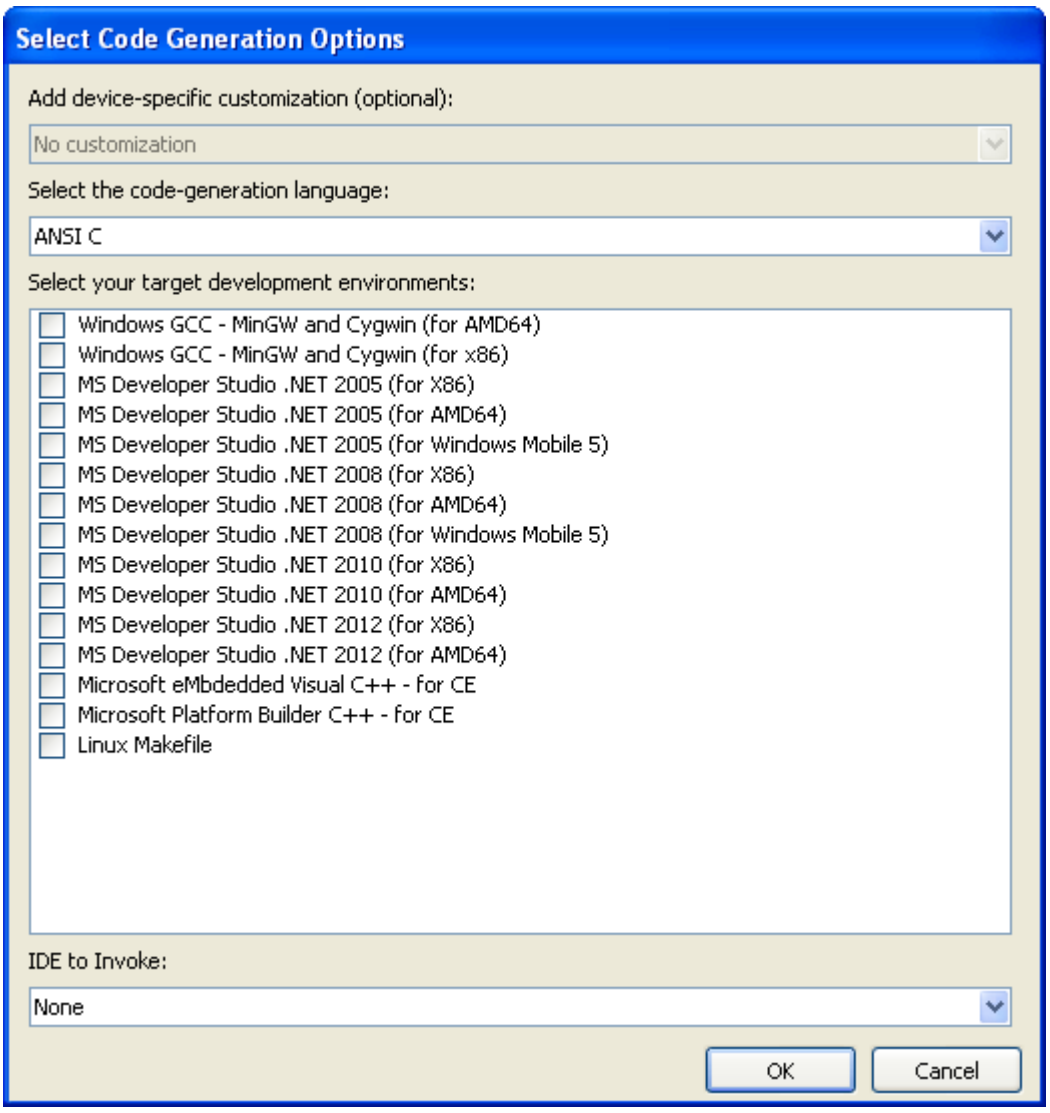
v. You can reset input and output pipes by pressing the **Reset Pipe** button for the selected pipe.

7. **Generate the skeletal driver code:**

a. Select to generate code either via the **Generate Code** toolbar icon or from the **Project | Generate Code** menu.

b. In the **Select Code Generation Options** dialogue box that will appear, choose the code language and development environment(s) for the generated code and select **Next** to generate the code.

Figure 5.10. Code Generation Options


Select Code Generation Options
Add device-specific customization (optional):
No customization
Select the code-generation language:
ANSI C
Select your target development environments:
Windows GCC - MinGW and Cygwin (for AMD64)
Windows GCC - MinGW and Cygwin (for x86)
MS Developer Studio .NET 2005 (for X86)
MS Developer Studio .NET 2005 (for AMD64)
MS Developer Studio .NET 2005 (for Windows Mobile 5)
MS Developer Studio .NET 2008 (for X86)
MS Developer Studio .NET 2008 (for AMD64)
MS Developer Studio .NET 2008 (for Windows Mobile 5)
MS Developer Studio .NET 2010 (for X86)
MS Developer Studio .NET 2010 (for AMD64)
MS Developer Studio .NET 2012 (for X86)
MS Developer Studio .NET 2012 (for AMD64)
Microsoft eMbdedded Visual C++ - for CE
Microsoft Platform Builder C++ - for CE
Linux Makefile
IDE to Invoke:
None
OK
Cancel

c. Save your project (if required) and click **OK** to open your development environment with the generated driver.

d. Close DriverWizard.

8. **Compile and run the generated code:**

   - Use this code as a starting point for your device driver. Modify where needed to perform your driver's specific functionality.

   - The source code DriverWizard creates can be compiled with any 32-bit compiler, and will run on all supported platforms without modification.

   For detailed compilation instructions, refer to Section 5.2.2.

5.2.1. Automatic Code Generation

After you have finished diagnosing your device and have ensured that it runs according to your specifications, you are ready to write your driver.

5.2.1.1. Generating the Code

Generate code by selecting this option either via DriverWizard's **Generate Code** toolbar icon or from the wizard's **Project | Generate Code** menu (see Section 5.2, Step 7). DriverWizard will generate the source code for your driver, and place it along with the project file (**xxx.wdp**, where "xxx" is the project name). The files are saved in a directory DriverWizard creates for every development environment and operating system selected in the code generation dialogue.

5.2.1.2. The Generated USB C Code

In the source code directory you now have a new **xxx_diag.c** source file (where **xxx** is the name you selected for your DriverWizard project). This file implements a diagnostic USB application, which demonstrates how to use WinDriver's USB API to locate and communicate with your USB device(s), including detection of Plug-and-Play events (device insertion/removal, etc.), performing read/write transfers on the pipes, resetting the pipes and changing the device's active alternate setting.
The generated application supports handling of multiple identical USB devices.

5.2.2. Compiling the Generated Code

5.2.2.1. Windows and Windows CE Compilation

As explained above, on Windows you can select to generate project, solution, and make files for the supported compilers and development environments — MS Visual Studio, Windows GCC (MinGW/Cygwin), MS eMbedded Visual C++, or MS Platform Builder.
For integrated development environments (IDEs), such as MS Visual Studio, you can also select to automatically invoke your selected IDE from the wizard. You can then proceed to immediately build and run the code from your selected IDE.

You can also build the generated code using any other compiler or development environment that supports the selected code language and target OS. Simply create a new project or make file for your selected compiler/environment, include the generated source files, and run the code.

- For Windows, the generated compiler/environment files are located under an **x86** directory — for 32-bit projects — or an **amd64** directory — for 64-bit projects.
- For Windows CE, note that the generated **Windows Mobile** code is targeted at the Windows Mobile 5.0/6.0 ARMV4I SDK.

5.2.2.2. Linux Compilation

Use the makefile that was created for you by DriverWizard in order to build the generated code using your favorite compiler, preferably GCC.

5.2.3. Bus Analyzer Integration — Ellisys Visual USB

DriverWizard provides native support for the Ellisys Explorer 200 USB analyzer on Windows XP and higher (32-bit only). This support enables you to:

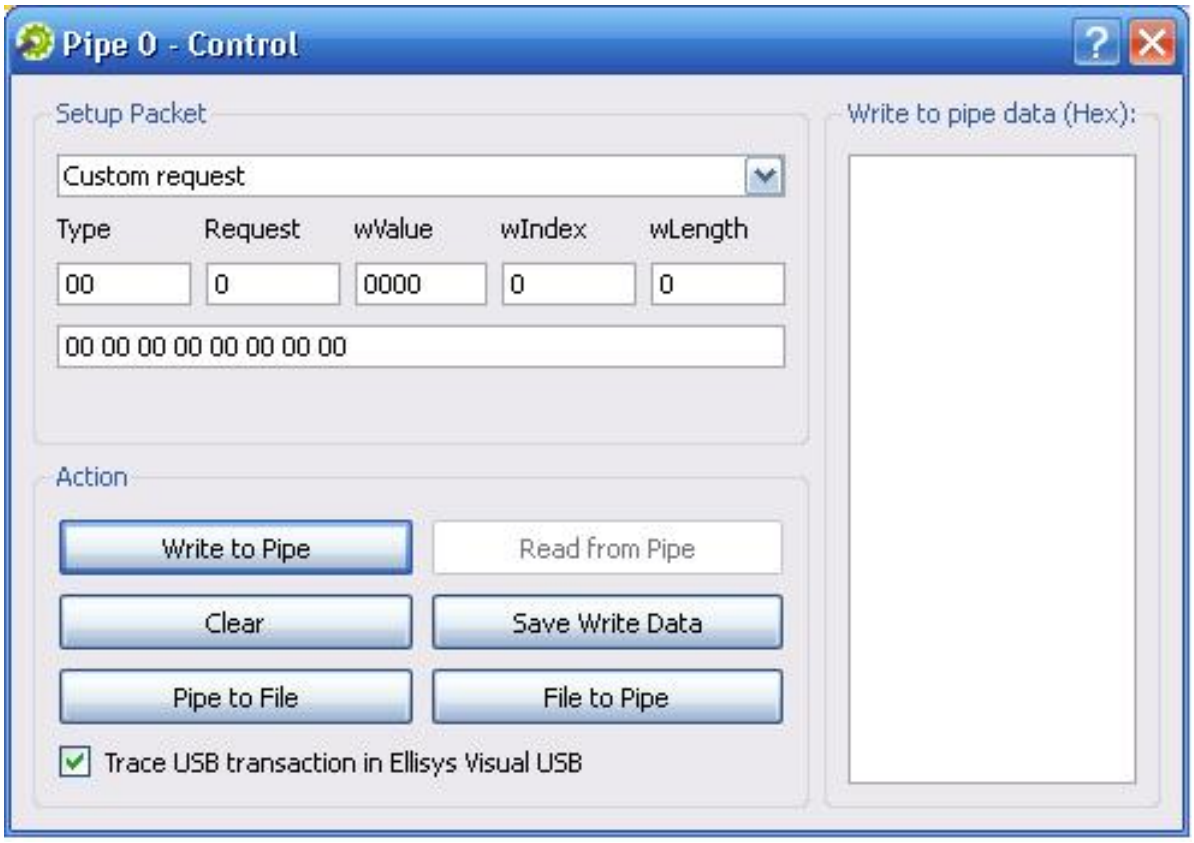
- Initiate USB traffic capture directly from DriverWizard.
- Capture discrete control transfers.

To capture USB traffic:

1. Select **Tools | Start USB Analyzer Capture** to start capturing USB data.
2. To finish the data capture, select **Tools | Stop USB Analyzer Capture**.
   A dialogue box will appear notifying you where DriverWizard stored the analyzer trace.
   Click **Yes** to run Ellisys's Visual Analyzer with the captured data.

To capture a discrete control trasfer check the **Trace USB transaction in Ellisys Visual USB** check box in the control transfers dialogue box.

Figure 5.11. Ellisys Visual USB Integration



Chapter 6
Developing a Driver

This chapter takes you through the WinDriver driver development cycle.

6.1. Using DriverWizard to Build a Device Driver

- Use DriverWizard to diagnose your device and verify that it operates as expected: View the device's configuration information, transfer data on its pipes, send standard requests to the control pipe, and reset the pipes.

- Use DriverWizard to generate skeletal code for your device in C, Visual Basic .NET, or C#. For more information about DriverWizard, refer to Chapter 5.

- Use any C or .NET compiler or development environment (depending on the code you created) to build the skeletal driver you need.
  WinDriver provides specific support for the following environments and compilers: MS Visual Studio, MS eMbedded Visual C++, MS Platform Builder C++, GCC, Windows GCC

That is all you need to do in order to create your user-mode driver.
For a detailed description of WinDriver's USB API, refer to Appendix B.
For more information regarding implementation of USB transfers with WinDriver, refer to Chapter 8.

6.2. Writing the Device Driver Without DriverWizard

There may be times when you choose to write your driver directly, without using DriverWizard. In such cases, either follow the steps outlined in this section to create a new driver project, or select a WinDriver sample that most closely resembles your target driver and modify it to suit your specific requirements.

6.2.1. Include the Required WinDriver Files

1. Include the relevant WinDriver header files in your driver project.
   All header files are found under the **WinDriver/include** directory.

   All WinDriver projects require the **windrvr.h** header file.

When using the `WDU_xxx` WinDriver USB API [B.2], include the **wdu_lib.h** header file; (this file already includes **windrvr.h**).

Include any other header file that provides APIs that you wish to use from your code (e.g., files from the **WinDriver/samples/shared** directory, which provide convenient diagnostics functions.)

2. Include the relevant header files from your source code: For example, to use the USB API from the **wdu_lib.h** header file, add the following line to the code:

```
#include "wdu_lib.h"
```

3. Link your code with the WDAPI library (Windows) / shared object (Linux):

   - For Windows: **WinDriver\lib\<CPU>\wdapi1150.lib**, where the **<CPU>** directory is either **x86** (32-bit binaries for x86 platforms), **amd64** (64-bit binaries for x64 platforms), or **amd64\x86** (32-bit binaries for x64 platforms [A.2]
   - For Windows CE: **WinDriver\lib\WINCE\<CPU>\wdapi1150.lib**
   - For Linux: From the **WinDriver/lib** directory — **libwdapi1150.so** or **libwdapi1150_32.so** (for 32-bit applications targeted at 64-bit platforms)
     Note: When using **libwdapi1150_32.so**, first create a copy of this file in a different directory and rename it to **libwdapi1150.so**, then link your code with the renamed file [A.2].

   You can also include the library's source files in your project instead of linking the project with the library. The C source files are located under the **WinDriver/src/wdapi** directory.

When linking your project with the WDAPI library/shared object, you will need to distribute this binary with your driver.
For Windows, get **wdapi1150.dll** / **wdapi1150_32.dll** (for 32-bit applications targeted at 64-bit platforms) from the **WinDriver\redist** directory.
For Linux, get **libwdapi1150.so** / **libwdapi1150_32.so** (for 32-bit applications targeted at 64-bit platforms) from the **WinDriver/lib** directory.

Note: On Windows and Linux, when using the DLL/shared object file for 32-bit applications on 64-bit platforms (**wdapi1150_32.dll** / **libwdapi1150_32.so**), rename the copy of the file in the distribution package, by removing the **_32** portion [A.2].
For detailed distribution instructions, refer to Chapter 10.

4. Add any other WinDriver source files that implement API that you which to use in your code (e.g., files from the **WinDriver/samples/shared** directory.)

6.2.2. Write Your Code

1. Call `WDU_Init()` [B.4.1] at the beginning of your program to initialize WinDriver for your USB device, and wait for the device-attach callback. The relevant device information will be provided in the attach callback.

2. Once the attach callback is received, you can start using one of the `WDU_Transfer()` [B.4.8.1] functions family to send and receive data.

3. To finish, call `WDU_Uninit()` [B.4.7] to unregister from the device.

6.2.3. Configure and Build Your Code

After including the required files and writing your code, make sure that the required build flags and environment variables are set, then build your code.

Before building your code, verify that the `WD_BASEDIR` environment variable is set to the location of the of the WinDriver installation directory.
On Windows, Windows CE, and Linux you can define the `WD_BASEDIR` environment variable globally — as explained in Chapter 4: For Windows — refer to the Windows WD_BASEDIR note in Section 4.2.1; for Windows CE — refer to Section 4.2.2.3; for Linux: refer to Section 4.2.3.2, Step 8.

6.3. Developing Your Driver on Windows CE Platforms

To use WinDriver to handle a Plug-and-Play device, you must first register the device with the WinDriver kernel module (**windrvr6.dll**).

To register the device with WinDriver, use either of the following methods:

- Modify the registry to identify your device and link it to **windrvr6.dll**. The registry can be modified by adding the relevant information to your **project.reg** file.

  - To identify the device by its vendor ID (<*VID*>) and product ID (<*PID*>) — as decimal values — add the following:

    ```
    [HKEY_LOCAL_MACHINE\Drivers\USB\LoadClients\<VID>_<PID>\Default\Default\WDR]
        "DLL"="windrvr6.dll"
    ```

  - To identify the device by its USB class (<*CLASS*>), subclass (<*SUBCLASS*>), and protocol (<*PROT*>) — as decimal values — add the following:

    ```
    [HKEY_LOCAL_MACHINE\Drivers\USB\LoadClients\Default\Default\<CLASS>_<SUBCLASS>_<PROT>\WDR]
        "DLL"="windrvr6.dll"
    ```

- Call `WDU_Init()` to identify the device by its vendor and product IDs and register it with WinDriver, before connecting the device to the computer .

For more information about the relevant registry settings, refer to *USB Driver Registry Settings* in the MSDN Library.

Chapter 7
Debugging Drivers

The following sections describe how to debug your hardware-access application code.

7.1. User-Mode Debugging

- Since WinDriver is accessed from the user mode, we recommend that you first debug your code using your standard debugging software.
- The Debug Monitor utility [7.2] logs debug messages from WinDriver's kernel-mode and user-mode APIs. You can also use WinDriver APIs to send your own debug messages to the Debug Monitor log.
- Use DriverWizard to validate your device's USB configuration and test the communication with the device.

7.2. Debug Monitor

Debug Monitor is a powerful graphical- and console-mode tool for monitoring all activities handled by the WinDriver kernel.
You can use this tool to monitor how each command sent to the kernel is executed.

In addition, WinDriver enables you to print your own debug messages to the Debug Monitor, using the `WD_DebugAdd`() function [B.6.6] or the high-level `PrintDbgMessage`() function [B.7.14].

The Debug Monitor comes in two versions:

- **wddebug_gui** [7.2.1] — a GUI version for Windows and Linux.
- **wddebug** [7.2.2] — a console-mode version for Windows, Windows CE, and Linux; on Windows CE, **wddebug** also supports GUI execution.

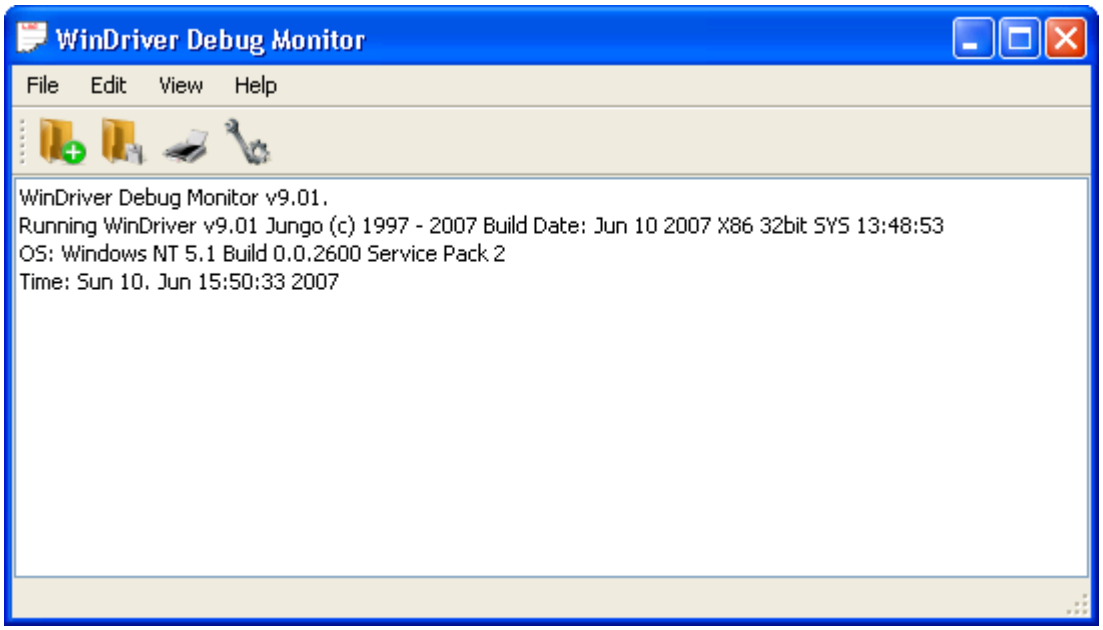
Both Debug Monitor versions are provided in the **WinDriver/util** directory.

7.2.1. The wddebug_gui Utility

wddebug_gui is a fully graphical (GUI) version of the Debug Monitor utility for Windows and Linux.

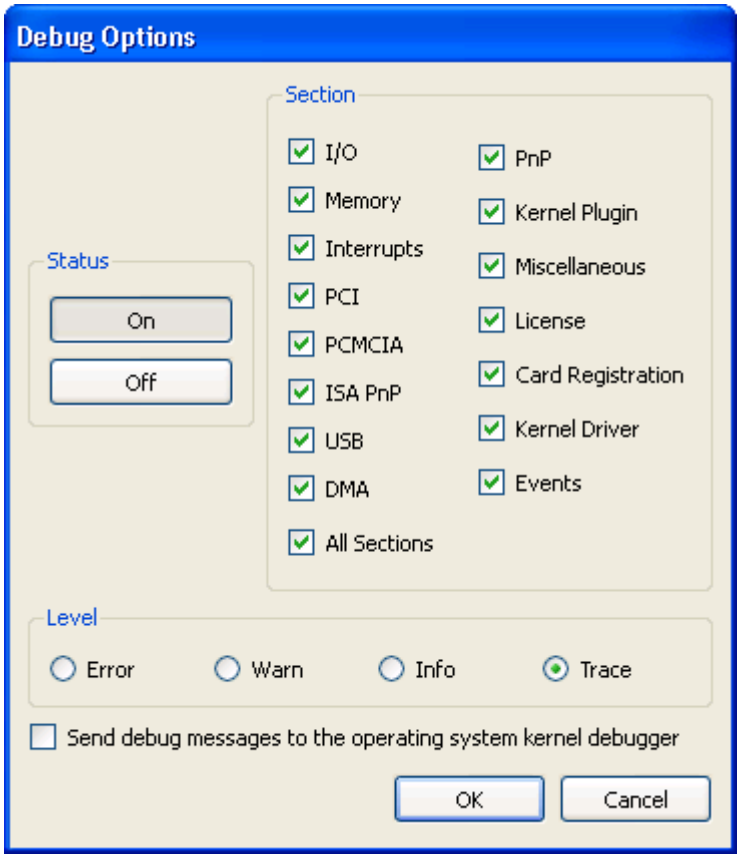
1. Run the Debug Monitor using either of the following methods:
   - Run **WinDriver/util/wddebug_gui**.
   - Run the Debug Monitor from DriverWizard's **Tools** menu.
   - On Windows, run **Start | Programs | WinDriver | Debug Monitor**.

Figure 7.1. Start Debug Monitor


WinDriver Debug Monitor
File
Edit
View
Help
WinDriver Debug Monitor v9.01.
Running WinDriver v9.01 Jungo (c) 1997 - 2007 Build Date: Jun 10 2007 X86 32bit SYS 13:48:53
OS: Windows NT 5.1 Build 0.0.2600 Service Pack 2
Time: Sun 10. Jun 15:50:33 2007

2. Set the Debug Monitor's status, trace level and debug sections information from the **Debug Options** dialogue, which is activated either from the Debug Monitor's **View | Debug Options** menu or the **Debug Options** toolbar button.

Figure 7.2. Debug Options



- **Status** — Set trace on or off.
- **Section** — Choose what part of the WinDriver API you would like to monitor.

  USB developers should select the **USB** section.

  Choose carefully those sections that you would like to monitor. Checking more options than necessary could result in an overflow of information, making it harder for you to locate your problem.

- **Level** — Choose the level of messages you want to see for the resources defined.
  - **Error** is the lowest trace level, resulting in minimum output to the screen.
  - **Trace** is the highest trace level, displaying every operation the WinDriver kernel performs.
- **Send debug messages to the operating system kernel debugger** —

  Select this option to send the debug messages received from the WinDriver kernel module to an external kernel debugger, in addition to the Debug Monitor.

On Windows Vista and higher, the first time that you enable this option you will need to restart the PC.

A free Windows kernel debugger, WinDbg, is distributed with the Windows Driver Kit (WDK) and is part of the Debugging Tools for Windows package, distributed via the Microsoft web site.

3. Once you have defined what you want to trace and on what level, click **OK** to close the **Debug Options** window.

4. Activate your application (step-by-step or in one run).

5. Watch the Debug Monitor log (or the kernel debugger log, if enabled) for errors or any unexpected messages.

7.2.1.1. Running wddebug_gui for a Renamed Driver

By default, **wddebug_gui** logs messages from the default WinDriver kernel module — **windrvr6.sys/.o/.ko**. However, you can also use **wddebug_gui** to log debug messages from a renamed version of this driver [11.2], by running **wddebug_gui** from the command line with the **`driver_name`** option: **`wddebug_gui <driver_name>`**.

The driver name should be set to the name of the driver file without the file's extension; e.g., **windrvr6**, not **windrvr6.sys** (on Windows) or **windrvr6.o** (on Linux).

For example, if you have renamed the default **windrvr6.sys** driver on Windows to **my_driver.sys**, you can log messages from your driver by running the Debug Monitor using the following command: **`wddebug_gui my_driver`**

7.2.2. The wddebug Utility

7.2.2.1. Console-Mode wddebug Execution

The **wddebug** version of the Debug Monitor utility can be executed as a console-mode application on all supported operating systems: Windows, Windows CE, and Linux. To use the console-mode Debug Monitor version, run **WinDriver/util/wddebug** in the manner explained below.

For console-mode execution on Windows CE, start a command window (**CMD.EXE**) on the Windows CE target, and then run the program **WDDEBUG.EXE** inside this shell. You can also execute **wddebug** via the Windows CE GUI, as explained in Section 7.2.2.2.

wddebug console-mode usage

```
wddebug [<driver_name>] [<command>] [<level>]
[<sections>]
```

The **wddebug** arguments must be provided in the order in which they appear in the usage statement above.

- **<driver_name>**: The name of the driver to which to apply the command.

  The driver name should be set to the name of the WinDriver kernel module — **windrvr6**, or a renamed version of this driver (refer to the explanation in Section 11.2).

The driver name should be set to the name of the driver file without the file's extension; for example, **windrvr6**, not **windrvr6.sys** (on Windows) or **windrvr6.o** (on Linux).

- **<command>**: The Debug Monitor command to execute:
  - Activation commands:
    - **on**: Turn the Debug Monitor on.
    - **off**: Turn the Debug Monitor off.
    - **dbg_on** : Redirect the debug messages from the Debug Monitor to a kernel debugger and turn the Debug Monitor on (if it was not already turned on).

On Windows Vista and higher, the first time that you enable this option you will need to restart the PC.

    - **dbg_off** : Stop redirecting debug messages from the Debug Monitor to a kernel debugger.

The **on** and **dbg_on** commands can be run together with the **<level>** and **<sections>** options, described below.

  - **dump**: Continuously display ('dump') debug information, until the user selects to stop.
  - **status**: Display information regarding the running **<driver_name>** driver, the current Debug Monitor status — including the active debug level and sections (when the Debug Monitor is on) — and the size of the debug messages buffer.
  - **help** : Display usage instructions.
  - None: You can run **wddebug** with no arguments, including no command. On platforms other than Windows CE, this is equivalent to running **wddebug help**. On Windows CE, running **wddebug** with no arguments activates the utility's Windows CE GUI version, as explained in Section 7.2.2.2.

The following options are applicable only to the **on** and **dbg_on** commands:

- **<level>**: The debug trace level to set. The level can be set to either of the following flags: **ERROR**, **WARN**, **INFO** or **TRACE**, where ERROR is the lowest trace level and TRACE is the highest level (displays all messages).
  The default debug trace level is TRACE.

- **<sections>**: The debug sections to set. The debug sections determine what part of the WinDriver API you would like to monitor.

  For a full list of all supported debug sections, run **wddebug help** to view the utility's usage instructions.
  The default debug sections flag is **ALL** — sets all the supported debug sections.

Usage Sequence

To log messages using **wddebug**, use the following sequence:

- Turn on the Debug Monitor by running **wddebug** with either the **on** command, or the **dbg_on** command — which redirects the debug messages to a kernel debugger before turning on the Debug Monitor.

  You can use the **level** and/or **sections** flags to set the debug level and/or sections for the log. If these options are not explicitly set, the default values will be used.

  You can also log messages from a renamed WinDriver driver by preceding the command with the name of the driver (see the **<driver_name>** option above). The default driver name is **windrvr6**.

- Run **wddebug** with the **dump** command to begin dumping debug messages to the command prompt. You can turn off the display of the debug messages, at any time, by following the instructions displayed in the command prompt.

- Run applications that use the driver, and view the debug messages as they are being logged to the command prompt/the kernel debugger.

- You can run **wddebug** with the **status** command, at any time while the Debug Monitor is on, to view the current debug level and sections, as well as information regarding the running <**driver_name**> kernel module.

- You can use **dbg_on** and **dbg_off** to toggle the redirection of debug messages to a kernel debugger at any time while the Debug Monitor is on.

- When you are ready, turn off the Debug Monitor by running **wddebug** with the **off** command.

You can also run **wddebug** with the **status** command while the Debug Monitor is turned off, to view information regarding the running <**driver_name**> driver.

Example

The following is an example of a typical **wddebug** usage sequence. Since no **<driver_name>** is set, the commands are applied to the default driver — **windrvr6**.

- Turn the Debug Monitor on with the highest trace level for all sections:
  wddebug on TRACE ALL

  Note: This is the same as running **'wddebug on TRACE'**, since **ALL** is the default debug sections option.

- Dump the debug messages continuously, until the user selects to stop:
  wddebug dump

- Use the driver and view the debug messages in the command prompt.

- Turn the Debug Monitor off:
  wddebug off

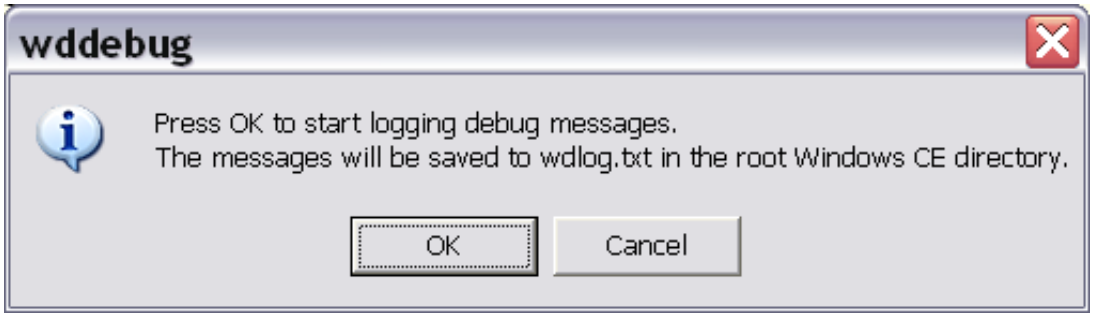
- Display usage instructions:
  wddebug help As explained above, on all platforms other than Windows CE, this is equivalent to running **wddebug** with no arguments.

7.2.2.2. Windows CE GUI wddebug Execution

On Windows CE, you can also log debug messages by running **wddebug** without any arguments. This method is designed to enable debug logging on Windows CE platforms that do not have a command-line prompt. On such platforms, you can activate debug logging by double-clicking the **wddebug** executable; this is equivalent to running the application with no arguments from a command-line prompt.

When executing **wddebug** without arguments, the user is informed, via a GUI message box, that log messages will be stored in a predetermined log file — **wdlog.txt** in the root Windows CE directory — and is given the option to cancel or continue.

Figure 7.3. wddebug Windows CE Start Log Message


wddebug
Press OK to start logging debug messages.
The messages will be saved to wdlog.txt in the root Windows CE directory.
OK
Cancel

If the user selects to continue, debug logging is turned on with a trace level of **TRACE** and debug sections **ALL**, and the Debug Monitor begins dumping debug messages to the **wdlog.txt** log file. The user can stop the logging and turn off debug logging, at any time, via a dedicated GUI message box.

Figure 7.4. wddebug Windows CE Stop Log Message

Chapter 8
USB Transfers

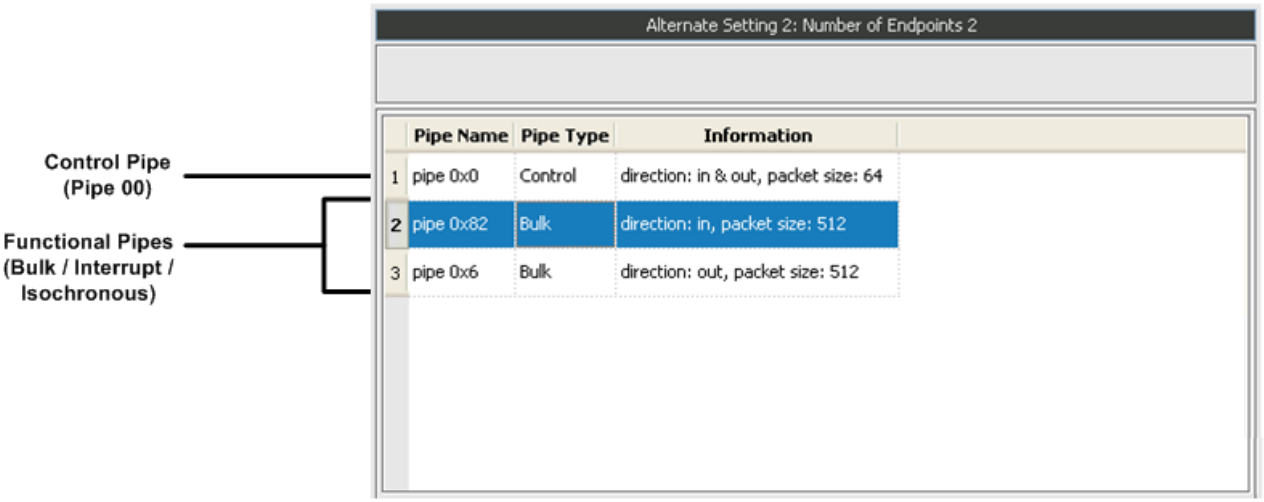
8.1. Overview

This chapter provides detailed information regarding implementation of USB transfers using WinDriver.

As explained in Section 3.5, the USB standard supports two kinds of data exchange between the host and the device — control exchange and functional data exchange.
The WinDriver APIs enable you to implement both control and functional data transfers.

Figure 8.1 demonstrates how a device's pipes are displayed in the DriverWizard utility, which enables you to perform transfers from a GUI environment.

Figure 8.1. USB Data Exchange


Control Pipe (Pipe 00)
Functional Pipes (Bulk / Interrupt / Isochronous)
Alternate Setting 2: Number of Endpoints 2
Pipe Name
Pipe Type
Information
1 pipe 0x0 Control direction: in & out, packet size: 64
2 pipe 0x82 Bulk direction: in, packet size: 512
3 pipe 0x6 Bulk direction: out, packet size: 512

Section 8.2 below provides detailed information regarding USB control transfers and how they can be implemented using WinDriver.

Section 8.3 describes the functional data transfer implementation options provided by WinDriver.

8.2. USB Control Transfers

8.2.1. USB Control Transfers Overview

8.2.1.1. Control Data Exchange

USB control exchange is used to determine device identification and configuration requirements and to configure a device, and can also be used for other device-specific purposes, including control of other pipes on the device.
Control exchange takes place via a control pipe — the default *pipe 0*, which always exists. The control transfer consists of a *setup stage* (in which a setup packet is sent from the host to the device), an optional *data stage* and a *status stage*.

8.2.1.2. More About the Control Transfer

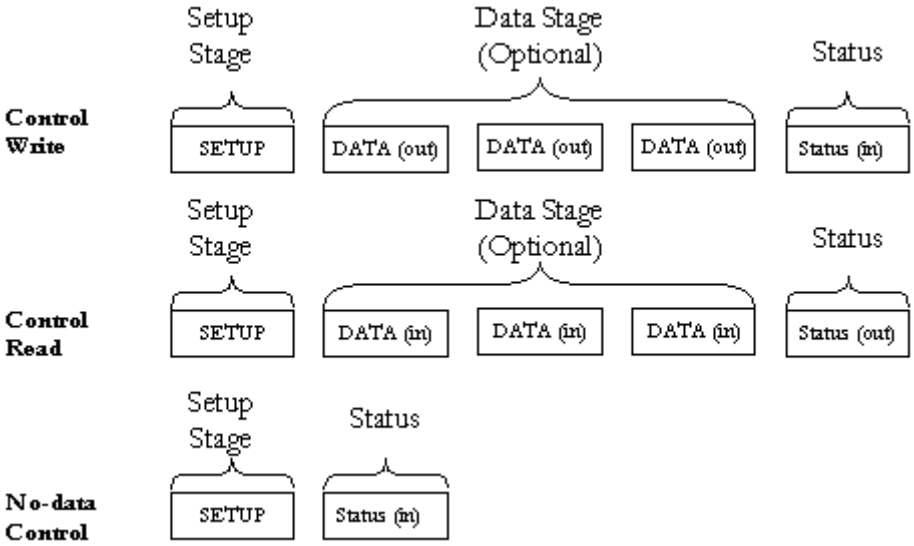
The control transaction always begins with a setup stage. The setup stage is followed by zero or more control data transactions (data stage) that carry the specific information for the requested operation, and finally a status transaction completes the control transfer by returning the status to the host.

During the setup stage, an 8-byte setup packet is used to transmit information to the control endpoint of the device (endpoint 0). The setup packet's format is defined by the USB specification.

A control transfer can be a read transaction or a write transaction. In a read transaction the setup packet indicates the characteristics and amount of data to be read from the device. In a write transaction the setup packet contains the command sent (written) to the device and the number of control data bytes that will be sent to the device in the data stage.

Refer to Figure 8.2 (taken from the USB specification) for a sequence of read and write transactions.
'(in)' indicates data flow from the device to the host.
'(out)' indicates data flow from the host to the device.

Figure 8.2. USB Read and Write



8.2.1.3. The Setup Packet

The setup packets (combined with the control data stage and the status stage) are used to configure and send commands to the device. Chapter 9 of the USB specification defines standard device requests. USB requests such as these are sent from the host to the device, using setup packets. The USB device is required to respond properly to these requests. In addition, each vendor may define device-specific setup packets to perform device-specific operations. The standard setup packets (standard USB device requests) are detailed below. The vendor's device-specific setup packets are detailed in the vendor's data book for each USB device.

8.2.1.4. USB Setup Packet Format

The table below shows the format of the USB setup packet. For more information, please refer to the USB specification at http://www.usb.org.

Byte	Field	Description
0	bmRequest Type	Bit 7: Request direction (0=Host to device — Out, 1=Device to host — In). Bits 5-6: Request type (0=standard, 1=class, 2=vendor, 3=reserved). Bits 0-4: Recipient (0=device, 1=interface, 2=endpoint,3=other).
1	bRequest	The actual request (see the Standard Device Request Codes table [8.2.1.5].
2	wValueL	A word-size value that varies according to the request. For example, in the `CLEAR_FEATURE` request the value is used to select the feature, in the `GET_DESCRIPTOR` request the value indicates the descriptor type and in the `SET_ADDRESS` request the value contains the device address.
3	wValueH	The upper byte of the `Value` word.

Byte	Field	Description
4	wIndexL	A word-size value that varies according to the request. The index is generally used to specify an endpoint or an interface.
5	wIndexH	The upper byte of the `Index` word.
6	wLengthL	A word-size value that indicates the number of bytes to be transferred if there is a data stage.
7	wLengthH	The upper byte of the `Length` word.

8.2.1.5. Standard Device Request Codes

The table below shows the standard device request codes.

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

8.2.1.6. Setup Packet Example

This example of a standard USB device request illustrates the setup packet format and its fields. The setup packet is in Hex format.

The following setup packet is for a control read transaction that retrieves the device descriptor from the USB device. The device descriptor includes information such as USB standard revision, vendor ID and product ID.

GET_DESCRIPTOR (Device) Setup Packet

80	06	00	01	00	00	12	00

Setup packet meaning:

Byte	Field	Value	Description
0	BmRequest Type	80	8h=1000b bit 7=1 -> direction of data is from device to host. 0h=0000b bits 0..1=00 -> the recipient is the device.
1	bRequest	06	The Request is GET_DESCRIPTOR.
2	wValueL	00	
3	wValueH	01	The descriptor type is device (values defined in USB spec).
4	wIndexL	00	The index is not relevant in this setup packet since there is only one device descriptor.
5	wIndexH	00	
6	wLengthL	12	Length of the data to be retrieved: 18(12h) bytes (this is the length of the device descriptor).
7	wLengthH	00	

In response, the device sends the device descriptor data. A device descriptor of the Cypress EZ-USB Integrated Circuit is provided as an example:

Byte No.	0	1	2	3	4	5	6	7	8	9	10
Content	12	01	00	01	ff	ff	ff	40	47	05	80

Byte No.	11	12	13	14	15	16	17
Content	00	01	00	00	00	00	01

As defined in the USB specification, byte 0 indicates the length of the descriptor, bytes 2-3 contain the USB specification release number, byte 7 is the maximum packet size for the control endpoint (endpoint 0), bytes 8-9 are the vendor ID, bytes 10-11 are the product ID, etc.

8.2.2. Performing Control Transfers with WinDriver

WinDriver allows you to easily send and receive control transfers on the control pipe (pipe 0), while using DriverWizard to test your device. You can either use the API generated by DriverWizard [5] for your hardware, or directly call the WinDriver `WDU_Transfer()` function [B.4.8.1] from within your application.

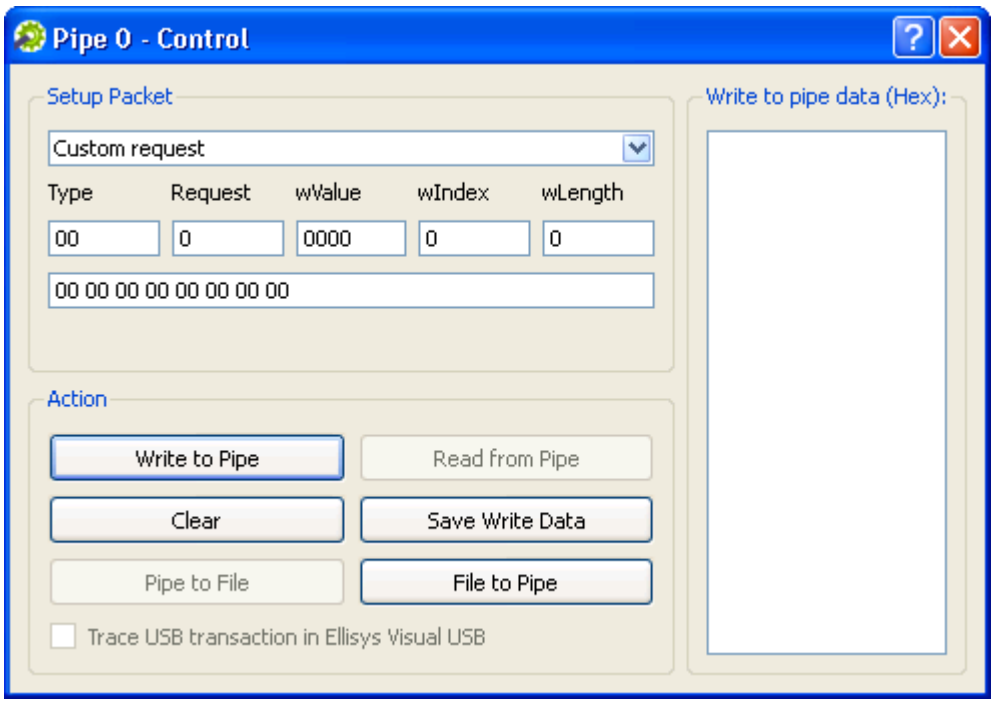
8.2.2.1. Control Transfers with DriverWizard

1. Choose **Pipe 0x0** and click the **Read / Write** button.

2. You can either enter a custom setup packet, or use a standard USB request.

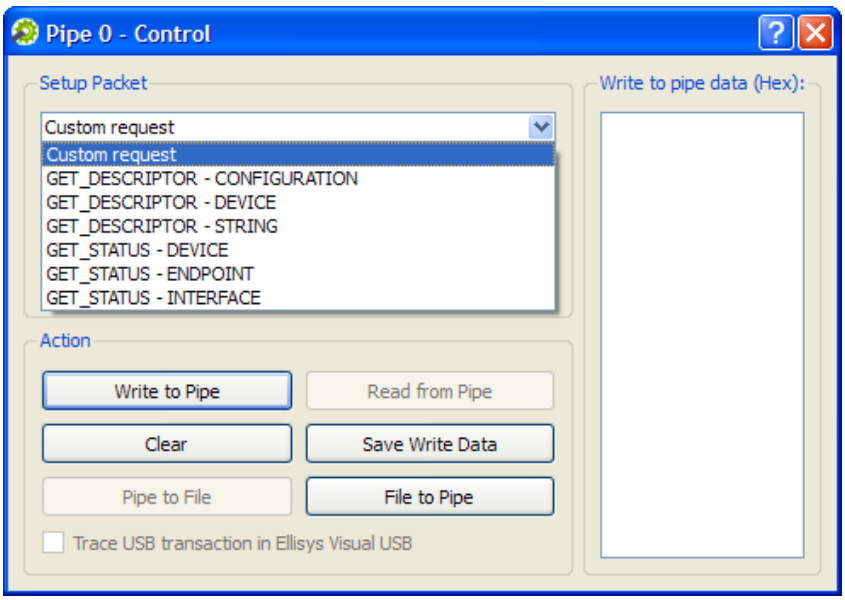
- For a custom request: enter the required setup packet fields. For a write transaction that includes a data stage, enter the data in the **Write to pipe data (Hex)** field. Click **Read From Pipe** or **Write To Pipe** according to the required transaction (see Figure 8.3).

Figure 8.3. Custom Request



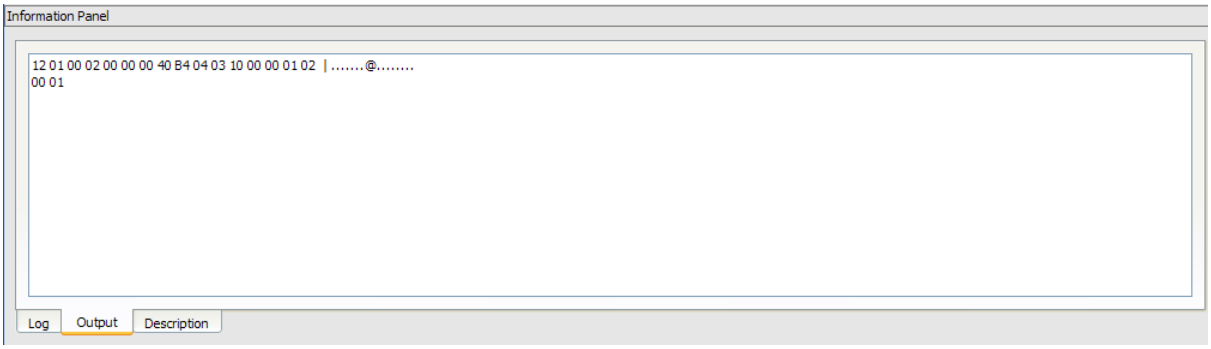
- For a standard USB request: select a USB request from the requests list, which includes requests such as **GET_DESCRIPTOR CONFIGURATION, GET_DESCRIPTOR DEVICE, GET_STATUS DEVICE**, etc. (see Figure 8.4). The description of the selected request will be displayed in the **Request Description** box on the right hand of the dialogue window.

Figure 8.4. Request List



3. The results of the transfer, such as the data that was read or a relevant error, are displayed in Driver Wizard's **Log** window.
   Figure 8.5, below, shows the contents of the **Log** window after a successful **GET_DESCRIPTOR DEVICE** request.

Figure 8.5. USB Request Log



8.2.2.2. Control Transfers with WinDriver API

To perform a read or write transaction on the control pipe, you can either use the API generated by DriverWizard for your hardware, or directly call the WinDriver `WDU_Transfer()` function [B.4.8.1] from within your application.

Fill the setup packet in the `BYTE SetupPacket[8]` array and call these functions to send setup packets on the control pipe (pipe 0) and to retrieve control and status data from the device.

- The following sample demonstrates how to fill the `SetupPacket[8]` variable with a `GET_DESCRIPTOR` setup packet:

```
setupPacket[0] = 0x80;   /* BmRequstType */
    setupPacket[1] = 0x6;    /* bRequest [0x6 == GET_DESCRIPTOR] */
    setupPacket[2] = 0;      /* wValue */
    setupPacket[3] = 0x1;    /* wValue [Descriptor Type: 0x1 == DEVICE] */
    setupPacket[4] = 0;      /* wIndex */
    setupPacket[5] = 0;      /* wIndex */
    setupPacket[6] = 0x12;   /* wLength [Size for the returned buffer] */
    setupPacket[7] = 0;      /* wLength */
```

- The following sample demonstrates how to send a setup packet to the control pipe (a `GET` instruction; the device will return the information requested in the `pBuffer` variable):

```
WDU_TransferDefaultPipe(hDev, TRUE, 0, pBuffer, dwSize,
       bytes_transferred, &setupPacket[0], 10000);
```

- The following sample demonstrates how to send a setup packet to the control pipe (a `SET` instruction):

```
WDU_TransferDefaultPipe(hDev, FALSE, 0, NULL, 0,
        bytes_transferred, &setupPacket[0], 10000);
```

For further information regarding `WDU_TransferDefaultPipe()`, refer to Section B.4.8.3.
For further information regarding `WDU_Transfer()`, refer to Section B.4.8.1.

8.3. Functional USB Data Transfers

8.3.1. Functional USB Data Transfers Overview

Functional USB data exchange is used to move data to and from the device. There are three types of USB data transfers: Bulk, Interrupt and Isochronous , which are described in detail in Sections 3.6.2–3.6.4 of the manual.

Functional USB data transfers can be implemented using two alternative methods: single-blocking transfers and streaming transfers, both supported by WinDriver, as explained in the following sections. The generated DriverWizard USB code [5.2.1] and the generic **WinDriver/util/usb_diag.exe** utility [1.9.2] (source code located under the **WinDriver/samples/usb_diag** directory) enable the user to select which type of transfer to perform.

8.3.2. Single-Blocking Transfers

In the single-blocking USB data transfer scheme, blocks of data are synchronously transferred (hence — "blocking") between the host and the device, per request from the host (hence — "single" transfers).

8.3.2.1. Performing Single-Blocking Transfers with WinDriver

WinDriver's **`WDU_Transfer()`** function, and the **`WDU_TransferBulk()`**, **`WDU_TransferIsoch()`**, and **`WDU_TransferInterrupt()`** convenience functions — all described in Section B.4.8 of the manual — enable you to easily impelment single-blocking USB data transfers.

You can also perform single-blocking transfers using the DriverWizard utility (which uses the `WDU_Transfer()` function), as demonstrated in Section 5.2 of the manual.

8.3.3. Streaming Data Transfers

In the streaming USB data transfer scheme, data is continuously streamed between the host and the device, using internal buffers allocated by the host driver — "streams".

Stream transfers allow for a sequential data flow between the host and the device, and can be used to reduce single-blocking transfer overhead, which may occur as a result of multiple function calls and context switches between user and kernel modes. This is especially relevant for devices with small data buffers, which might, for example, overwrite data before the host is able to read it, due to a gap in the data flow between the host and device.

8.3.3.1. Performing Streaming with WinDriver

WinDriver's **WDU_StreamXXX()** functions, described in Section B.4.9 of the manual, enable you to impelment USB streaming data transfers. Note: These functions are currently supported on Windows and Windows CE.

To begin performing stream transfers, call the **WDU_StreamOpen()** function [B.4.9.1]. When this function is called, WinDriver creates a new stream object for the specified data pipe. You can open a stream for any pipe except for the control pipe (pipe 0). The stream's data transfer direction — read/write — is derived from the direction of its pipe.
WinDriver supports both blocking and non-blocking stream transfers. The open function's `fBlocking` parameter indicates which type of transfer to perform (see explanation below). Streams that perform blocking transfers will henceforth be referred to as "blocking streams", and streams that perform non-blocking transfers will be referred to as "non-blocking streams".
The function's `dwRxTxTimeout` parameter indicates the desired timeout period for transfers between the stream and the device.

After opening a stream, call **WDU_StreamStart()** [B.4.9.2] to begin data transfers between the stream's data buffer and the device.
In the case of a read stream, the driver will constantly read data from the device into the stream's buffer, in blocks of a pre-defined size (as set in the `dwRxSize` parameter of the `WDU_StreamOpen`() function [B.4.9.1]. In the case of a write stream, the driver will constantly check for data in the stream's data buffer and write any data that is found to the device.

To read data from a read stream to the user-mode host application, call **WDU_StreamRead()** [B.4.9.3].
In case of a blocking stream, the read function blocks until the entire amount of data requested by the application is transferred from the stream to the application, or until the stream's attempt to read data from the device times out.
In the case of a non-blocking stream, the function transfers to the application as much of the requested data as possible, subject to the amount of data currently available in the stream's data buffer, and returns immediately.

To write data from the user-mode host application to a write the stream, call **WDU_StreamWrite()** [B.4.9.4].
In case of a blocking stream, the function blocks until the entire data is written to the stream, or until the stream's attempt to write data to the device times out.
In the case of a non-blocking stream, the function writes as much of the write data as currently possible to the stream, and returns immediately.

For both blocking and non-blocking transfers, the read/write function returns the amount of bytes actually transferred between the stream and the calling application within an output parameter — `*pdwBytesRead` [B.4.9.3] / `*pdwBytesWritten` [B.4.9.4].

You can flush an active stream at any time by calling the **WDU_StreamFlush()** function [B.4.9.5], which writes the entire contents of the stream's data buffer to the device (for a write stream), and blocks until all pending I/O for the stream is handled.
You can flush both blocking and non-blocking streams.

You can call `WDU_StreamGetStatus`() [B.4.9.6] for any open stream in order to get the stream's current status information.

To stop the data streaming between an active stream and the device, call **`WDU_StreamStop()`** [B.4.9.7]. In the case of a write stream, the function flushes the stream — i.e., writes its contents to the device — before stopping it.
An open stream can be stopped and restarted at any time until it is closed.

To close an open stream, call **`WDU_StreamClose()`** [B.4.9.8].
The function stops the stream, including flushing its data to the device (in the case of a write stream), before closing it.
Note: Each call to `WDU_StreamOpen`() must have a matching call to `WDU_StreamClose`() later on in the code in order to perform the necessary cleanup.

Chapter 9
Dynamically Loading Your Driver

9.1. Why Do You Need a Dynamically Loadable Driver?

When adding a new driver, you may be required to reboot the system in order for it to load your new driver into the system. WinDriver is a dynamically loadable driver, which enables your customers to start your application immediately after installing it, without the need for reboot.

To successfully unload your driver, make sure that there are no open handles to the WinDriver service (**windrvr6.sys** or your renamed driver (refer to Section 11.2), and that there are no connected and enabled Plug-and-Play devices that are registered with this service.

9.2. Windows Dynamic Driver Loading

Windows XP and higher uses Windows Driver Model (WDM) drivers [2.3.1]: Files with the extension ***.sys** (e.g., **windrvr6.sys**).
WDM drivers are installed via the installation of an INF file (see below).

The WinDriver Windows kernel module — **windrvr6.sys** — is a fully WDM driver, which can be installed using the **wdreg** utility, as explained in the following sections.

9.2.1. The wdreg Utility

WinDriver provides a utility for dynamically loading and unloading your driver, which replaces the slower manual process using Windows' Device Manager (which can still be used for the device INF). This utility is provided in two forms: **wdreg** and **wdreg_gui**. Both versions can be found in the **WinDriver\util** directory, can be run from the command line, and provide the same functionality. The difference is that **wdreg_gui** displays installation messages graphically, while **wdreg** displays them in console mode.

This section describes the use of **wdreg**/ **wdreg_gui** on Windows operating systems.

1. **wdreg** is dependent on the Driver Install Frameworks API (**DIFxAPI**) DLL — **difxapi.dll**, unless when run with the **`-compat`** option (described below). **difxapi.dll** is provided under the **WinDriver\util** directory.

2. The explanations and examples below refer to **wdreg**, but any references to **wdreg** can be replaced with **wdreg_gui**.

9.2.1.1. Overview

This section explains how to use the **wdreg** utility to install the WDM **windrvr6.sys** driver, or to install INF files that register USB devices to work with this driver, on Windows.

You can rename the **windrvr6.sys** kernel module and modify your device INF file to register with your renamed driver, as explained in Section 11.2.1. To install your modified INF files using **wdreg**, simply replace any references to **windrvr6** below with the name of your new driver.

Usage: The **wdreg** utility can be used in two ways as demonstrated below:

1. **`wdreg -inf <filename> [-silent] [-log <logfile>] [install | preinstall | uninstall | enable | disable]`**

2. **`wdreg -rescan <enumerator> [-silent] [-log <logfile>]`**

- OPTIONS
  wdreg supports several basic OPTIONS from which you can choose one, some, or none:

  - **-inf** — The path of the INF file to be dynamically installed.

  - **-rescan <enumerator>** — Rescan enumerator (ROOT, USB, etc.) for hardware changes. Only one enumerator can be specified.

  - **-silent** — Suppress display of all messages (optional).

  - **-log <logfile>** — Log all messages to the specified file (optional).

  - **-compat** — Use the traditional **SetupDi** API instead of the newer Driver Install Frameworks API (**DIFxAPI**).

- ACTIONS
  wdreg supports several basic ACTIONS:

  - **install** — Installs the INF file, copies the relevant files to their target locations, and dynamically loads the driver specified in the INF file name by replacing the older version (if needed).

  - **preinstall** Pre-installs the INF file for a non-present device.

- **uninstall** — Removes your driver from the registry so that it will not load on next boot (see note below).
- **enable** — Enables your driver.
- **disable** — Disables your driver, i.e., dynamically unloads it, but the driver will reload after system boot (see note below).

To successfully disable/uninstall your driver, make sure that there are no open handles to the WinDriver service (**windrvr6.sys** or your renamed driver (refer to Section 11.2), and that there are no connected and enabled Plug-and-Play devices that are registered with this service.

9.2.2. Dynamically Loading/Unloading windrvr6.sys INF Files

When using WinDriver, you develop a user-mode application that controls and accesses your hardware by using the generic **windrvr6.sys** driver (WinDriver's kernel module). Therefore, you might want to dynamically load and unload the driver **windrvr6.sys** — which you can do using **wdreg**.
In addition, in WDM-compatible operating systems, you also need to dynamically load INF files for your Plug-and-Play devices. **wdreg** enables you to do so automatically on Windows.
This section includes **wdreg** usage examples, which are based on the detailed description of **wdreg** contained in the previous section.

- To load **windrvr6.inf** and start the **windrvr6.sys** service —
  `wdreg -inf <path to windrvr6.inf> install`
- To load an INF file named **device.inf**, located in the **c:\tmp** directory —
  `wdreg -inf c:\tmp\device.inf install`

  You can replace the **`install`** option in the example above with **`preinstall`** to pre-install the device INF file for a device that is not currently connected to the PC.

If the installation fails with an **`ERROR_FILE_NOT_FOUND`** error, inspect the Windows registry to see if the **`RunOnce`** key exists in **`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`**. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the **`RunOnce`** key is missing, create it; then try installing the INF file again.

To unload the driver/INF file, use the same commands, but simply replace **`install`** in the examples above with **`uninstall`**.

9.3. Linux Dynamic Driver Loading

The following commands must be executed with root privileges.

- To dynamically load WinDriver, run the following command:
  `# <path to wdreg> windrvr6`
- To dynamically unload WinDriver, run the following command:
  `# /sbin/modprobe -r windrvr6`.

wdreg is provided in the **WinDriver/util** directory.

To automatically load WinDriver on each boot, add the following line to the target's Linux boot file (for example, **/etc/rc.local**):

```
<path to wdreg> windrvr6
```

9.4. Windows CE Dynamic Driver Loading

The **WinDriver\redist\Windows_Mobile_5_ARMV4I\wdreg.exe** utility can be used for loading the WinDriver kernel module (**windrvr6.dll**) on a Windows CE platform.

On many versions of Windows CE, the operating system's security scheme prevents the loading of unsigned drivers at boot time, therefore the WinDriver kernel module has to be reloaded after boot. To load WinDriver on the target Windows CE platform every time the OS is started, copy the **wdreg.exe** utility to the **Windows\StartUp** directory on the target PC.

The source code of the Windows CE **wdreg.exe** utility is available under the **WinDriver\samples\wince_install\wdreg** directory on the development PC.

Chapter 10
Distributing Your Driver

Read this chapter in the final stages of driver development. It will guide you in preparing your driver for distribution.

10.1. Getting a Valid License for WinDriver

To purchase a WinDriver license, complete the **WinDriver/docs/order.pdf** order form and fax or email it to Jungo. Complete details are included on the order form. Alternatively, you can order WinDriver online. For more details, visit our web site: http://www.jungo.com.

In order to install the registered version of WinDriver and to activate driver code that you have developed during the evaluation period on the development machine, please follow the installation instructions found in Section 4.2 above.

10.2. Windows Driver Distribution

- All references to **wdreg** in this section can be replaced with **wdreg_gui**, which offers the same functionality as **wdreg** but displays GUI messages instead of console-mode messages.

- If you have renamed the WinDriver kernel module (**windrvr6.sys**), as explained in Section 11.2, replace the relevant **windrvr6** references with the name of your driver, and replace references to the **WinDriver\redist** directory with the path to the directory that contains your modified installation files. For example, when using the generated DriverWizard renamed driver files for your driver project, as explained in Section 11.2.1, you can replace references to the **WinDriver\redist** directory with references to the generated **xxx_installation\redist** directory (where **xxx** is the name of your generated driver project). Note also the option to simplify the installation using the generated DriverWizard **xxx_install.bat** script and the copies of the **WinDriver\util** installation files in the generated **xxx_installation\redist** directory, as explained in Section 11.2.1.

- If you have created new INF and/or catalog files for your driver, replace the references to the original WinDriver INF files and/or to the **wd1150.cat** catalog file with the names of your new files (see the file renaming information in Sections 11.2.1 and 11.3.2).

Distributing the driver you created is a multi-step process. First, create a distribution package that includes all the files required for the installation of the driver on the target computer. Second, install the driver on the target machine. This involves installing **windrvr6.sys** and **windrvr6.inf**, and installing the specific INF file for your device.

Finally, you need to install and execute the hardware-control application that you developed with WinDriver. These steps can be performed using **wdreg** utility.

10.2.1. Preparing the Distribution Package

Prepare a distribution package that includes the following files.

If you wish to distribute drivers for both 32-bit and 64-bit target platforms, you must prepare separate distribution packages for each platform. The required files for each package are provided in the WinDriver installation directory for the respective platform.

- Your hardware-control application/DLL.
- **windrvr6.sys**.
  Get this file from the **WinDriver\redist** directory of the WinDriver package.
- **windrvr6.inf**.
  Get this file from the **WinDriver\redist** directory of the WinDriver package.
- **wd1150.cat**.
  Get this file from the **WinDriver\redist** directory of the WinDriver package.
- **wdapi1150.dll** (for distribution of 32-bit binaries to 32-bit target platforms or for distribution of 64-bit binaries to 64-bit platforms) or **wdapi1150_32.dll** (for distribution of 32-bit binaries to 64-bit platforms [A.2].
  Get this file from the **WinDriver\redist** directory of the WinDriver package.
- **difxapi.dll** (required by the **wdreg.exe** utility [9.2.1]).
  Get this file from the **WinDriver\util** directory of the WinDriver package.
- An INF file for your device.
  You can generate this file with DriverWizard, as explained in Section 5.2.

10.2.2. Installing Your Driver on the Target Computer

Driver installation on Windows requires administrator privileges.

Follow the instructions below in the order specified to properly install your driver on the target computer:

- **Preliminary Steps:**
  To successfully install your driver, make sure that there are no open handles to the WinDriver service (**windrvr6.sys** or your renamed driver (refer to Section 11.2), and that there are no connected and enabled Plug-and-Play devices that are registered with this service. This is relevant, for example, when upgrading the version of the driver (for WinDriver v6.0.0 and above; earlier versions used a different module name). If the service is being used, attempts to install the new driver using **wdreg** will fail. You can disable or uninstall connected devices from the Device Manager (**Properties | Disable/Uninstall**) or using **wdreg**, or otherwise physically disconnect the device(s) from the PC.

- **Install WinDriver's kernel module:**

  1. Copy **windrvr6.sys**, **windrvr6.inf**, and **wd1150.cat** to the same directory.

     wd1150.cat contains the driver's Authenticode digital signature. To maintain the signature's validity this file must be found in the same installation directory as the **windrvr6.inf** file. If you select to distribute the catalog and INF files in different directories, or make any changes to these files or to any other files referred to by the catalog file (such as **windrvr6.sys**), you will need to do either of the following:
     - Create a new catalog file and re-sign the driver using this file.
     - Comment-out or remove the following line in the **windrvr6.inf** file:
       `CatalogFile=wd1150.cat`
       and do not include the catalog file in your driver distribution. However, note that this option invalidates the driver's digital signature.

     For more information regarding driver digital signing and certification and the signing of your WinDriver-based driver, refer to Section 11.3 of the manual.

  2. Use the utility **wdreg** to install WinDriver's kernel module on the target computer:
     `wdreg -inf <path to windrvr6.inf> install`

     For example, if **windrvr6.inf** and **windrvr6.sys** are in the **d:\MyDevice** directory on the target computer, the command should be:
     `wdreg -inf d:\MyDevice\windrvr6.inf install`

     You can find the executable of **wdreg** in the WinDriver package under the **WinDriver\util** directory. For a general description of this utility and its usage, please refer to Chapter 9.

     - **wdreg** is dependent on the **difxapi.dll** DLL.
     - **wdreg** is an interactive utility. If it fails, it will display a message instructing the user how to overcome the problem. In some cases the user may be asked to reboot the computer.

When distributing your driver, you should attempt to ensure that the installation does not overwrite a newer version of **windrvr6.sys** with an older version of the file in Windows drivers directory (**%windir%\system32\drivers**) — for example, by configuring your installation program (if you are using one) or your INF file so that the installer automatically compares the time stamp on these two files and does not overwrite a newer version with an older one. The provided **windrvr6.inf** file uses the `COPYFLG_NO_VERSION_DIALOG` directive, which is designed to avoid overwriting a file in the destination directory with the source file if the existing file is newer than the source file. There is also a similar `COPYFLG_OVERWRITE_OLDER_ONLY` INF directive that is designed to ensure that the source file is copied to the destination directory only if the destination file is superseded by a newer version. **Note**, however, that both of these INF directives are irrelevant to digitally signed drivers. As explained in the Microsoft *INF CopyFiles Directive* documentation — http://msdn.microsoft.com/en-us/library/ff546346%28v=vs.85%29.aspx — if a driver package is digitally signed, Windows installs the package as a whole and does not selectively omit files in the package based on other versions already present on the computer. The **windrvr6.sys** driver provided by Jungo is digitally signed (refer to Section 11.3 for more information).

- **Install the INF file for your device** (registering your Plug-and-Play device with **windrvr6.sys**):

  Run the utility **wdreg** with the **`install`** command to automatically install the INF file and update Windows Device Manager:

  ```
  wdreg -inf <path to your INF file> install
  ```

  You can also use the **wdreg** utility's **`preinstall`** command to pre-install an INF file for a device that is not currently connected to the PC:

  ```
  wdreg -inf <path to your INF file> preinstall
  ```

If the installation fails with an **`ERROR_FILE_NOT_FOUND`** error, inspect the Windows registry to see if the **`RunOnce`** key exists in **`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`**. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the **`RunOnce`** key is missing, create it; then try installing the INF file again.

- **Install wdapi1150.dll:**
  If your hardware-control application/DLL uses **wdapi1150.dll** (as is the case for the sample and generated DriverWizard WinDriver projects), copy this DLL to the target's **%windir%\system32** directory.
  If you are distributing a 32-bit application/DLL to a target 64-bit platform [A.2], rename **wdapi1150_32.dll** in your distribution package to **wdapi1150.dll**, and copy the renamed file to the target's **%windir%\sysWOW64** directory.

If you attempt to write a 32-bit installation program that installs a 64-bit program, and therefore copies the 64-bit **wdapi1150.dll** DLL to the **%windir%\system32** directory, you may find that the file is actually copied to the 32-bit **%windir%\sysWOW64** directory. The reason for this is that Windows x64 platforms translate references to 64-bit directories from 32-bit commands into references to 32-bit directories. You can avoid the problem by using 64-bit commands to perform the necessary installation steps from your 32-bit installation program. The **system64.exe** program, provided in the **WinDriver\redist** directory of the Windows x64 WinDriver distributions, enables you to do this.

- **Install your hardware-control application/DLL**: Copy your hardware-control application/ DLL to the target and run it!

10.3. Windows CE Driver Distribution

10.3.1. Distribution to New Windows CE Platforms

The following instructions apply to platform developers who build Windows CE kernel images using Windows CE Platform Builder or using MS Visual Studio 2005/2008 with the appropriate Windows CE plugin. The instructions use the notation **'Windows CE IDE'** to refer to either of these platforms.

To distribute the driver you developed with WinDriver to a new target Windows CE platform, follow these steps:

1. If you have not already done so, modify the project registry file — **WinDriver\samples\wince_install\project_wd.reg** — to add an entry for your target device.

2. Compile your Windows CE platform (Sysgen stage).

3. Integrate the driver into your platform:

   a. Run the Windows CE IDE and open your platform.

   b. Select **Open Release Directory** from the **Build** menu.

   c. Copy the WinDriver CE kernel file — **WinDriver\redist\<TARGET_CPU>\windrvr6.dll** — to the **%_FLATRELEASEDIR%** subdirectory on the target development platform (should be the current directory in the new command window).

   d. Append the contents of **WinDriver\samples\wince_install\project_wd.reg** to the **%_FLATRELEASEDIR%\project.reg** registry file.

e. Copy the contents of the **WinDriver\samples\wince_install\project_wd.bib** file to the FILES section of the binary image builder file — **%_FLATRELEASEDIR%\project.bib**. Then uncomment the line that matches the target platform (see the "TODO" comments in the copied text).

This step is only necessary if you want the WinDriver CE kernel file (**windrvr6.dll**) to be a permanent part of the Windows CE image (**NK.BIN**), which is the case if you select to transfer the file to your target platform using a boot disk. If you prefer to have the file **windrvr6.dll** loaded on demand via the CESH/PPSH services, you do not need to perform this step until you build a permanent kernel.

4. Select **Make Run-Time Image** from the **Build** menu, and name the new image **NK.BIN**.

5. Download your new kernel to the target platform and initialize it either by selecting **Attach Device** from the **Target** menu, or by using a boot disk. For Windows CE 4.x, the menu is called **Download/Initialize** rather than **Attach Device**.

6. Restart your target CE platform. The WinDriver CE kernel will automatically load.

7. Install your hardware-control application/DLL on the target.
   If your hardware-control application/DLL uses **wdapi1150.dll** (as is the case for the sample and generated DriverWizard WinDriver projects), also copy this DLL from the **WinDriver\redist\WINCE\<TARGET_CPU>** directory on the Windows host development PC to the target's **Windows** directory.

10.3.2. Distribution to Windows CE Computers

Unless otherwise specified, 'Windows CE' references in this section include all supported Windows CE platforms, including Windows Mobile.

1. Copy WinDriver's kernel module — **windrvr6.dll** — from the **WinDriver\redist\WINCE\<TARGET_CPU>** directory on the Windows host development PC to the **Windows** directory on your target Windows CE platform.

2. Add WinDriver to the list of device drivers Windows CE loads on boot:

   - Modify the registry according to the entries documented in the file **WinDriver\samples\wince_install\project_wd.reg**. This can be done using the Windows CE Pocket Registry Editor on the hand-held CE computer, or by using the Remote CE Registry Editor Tool supplied with MS eMbedded Visual C++ or MS Visual Studio 2005/2008. Note that in order to use the Remote CE Registry Editor tool you will need to have Windows CE Services installed on your Windows host platform.

- On many versions of Windows CE, the operating system's security scheme prevents the loading of unsigned drivers at boot time, therefore the WinDriver kernel module has to be reloaded after boot. To load WinDriver on the target Windows CE platform every time the OS is started, copy the **WinDriver\redist\Windows_Mobile_5_ARMV4I\wdreg.exe** utility to the **Windows\StartUp** directory on the target PC.

3. Restart your target CE computer. The WinDriver CE kernel will automatically load. You will have to do a warm reset rather than just suspend/resume (use the reset or power button on your target CE computer).

4. Install your hardware-control application/DLL on the target.
   If your hardware-control application/DLL uses **wdapi1150.dll** (as is the case for the sample and generated DriverWizard WinDriver projects), also copy this DLL from the **WinDriver\redist\WINCE\<TARGET_CPU>** directory on the development PC to the target's **Windows** directory.

10.4. Linux Driver Distribution

To distribute your driver, prepare a distribution package containing the required files — as outlined in Section 10.4.1 — and then build and install the required driver components on the target — as outlined in Sections 10.4.2–10.4.3.

- If you have renamed the WinDriver driver modules [11.2], replace references to **windrvr6** in the following instructions with the name of your renamed driver module.
- It is recommended that you supply an installation shell script to automate the build and installation processes on the target.

10.4.1. Preparing the Distribution Package

Prepare a distribution package containing the required files, as described in this section.

- If you wish to distribute drivers for both 32-bit and 64-bit target platforms, you must prepare separate distribution packages for each platform. The required files for each package are provided in the WinDriver installation directory for the respective platform.
- In the following instructions, **<source_dir>** represents the source directory from which to copy the distribution files. The default source directory is your WinDriver installation directory. However, if you have renamed the WinDriver driver modules [11.2], the source directory is a directory containing modified files for compiling and installing the renamed drivers; when using DriverWizard to generate the driver code, the source directory for the renamed driver is the generated **xxx_installation** directory, where **xxx** is the name of your generated driver project (see Section 11.2.2, Step 1).

10.4.1.1. Kernel Module Components

WinDriver uses two kernel modules: the main WinDriver driver module, which implements the WinDriver API — **windrvr6.o/.ko** — and a driver module that implements the USB functionality — **windrvr6_usb.o/.ko**.

Your kernel driver modules cannot be distributed as-is; they must be recompiled on each target machine, to match the kernel version on the target. This is due to the following reason: The Linux kernel is continuously under development, and kernel data structures are subject to frequent changes. To support such a dynamic development environment, and still have kernel stability, the Linux kernel developers decided that kernel modules must be compiled with header files identical to those with which the kernel itself was compiled. They enforce this by including a version number in the kernel header files, which is checked against the version number encoded into the kernel. This forces Linux driver developers to support recompilation of their driver with the target system's kernel version.

Following is a list of the components you need to distribute to enable compilation of your kernel driver modules on the target machine.

It is recommended that you copy the files to subdirectories in the distribution directory that match the source subdirectories, such as **redist** and **include**, except where otherwise specified. If you select not do so, you will need to modify the file paths in the configuration scripts and related makefile templates, to match the location of the files in your distribution directory.

- From the <source_dir>**/include** directory, copy **windrvr.h**, **wd_ver.h**, and **windrvr_usb.h** — header files required for building the kernel modules on the target.
- From the **<WinDriver installation directory>/util** directory (or from the generated DriverWizard **xxx_installation/redist** directory), copy **wdreg** — a script for loading the WinDriver kernel driver modules (see Section 9.3) — to the **redist** distribution directory.
- From the <source_dir>/**redist** directory, unless where otherwise specified, copy the following files:
  - **setup_inst_dir** — a script for installing the WinDriver driver modules, using **wdreg** (see above).
  - **linux_wrappers.c/.h** — wrapper library source code files that bind the kernel module to the Linux kernel.
  - **linux_common.h** and **wdusb_interface.h** — header files required for building the kernel modules on the target.
  - **wdusb_linux.c** — source file used by WinDriver to utilize the USB stack.

- The compiled object code for building the WinDriver kernel driver modules —
  - **windrvr_gcc_v3.a** — for GCC v3.x.x compilation
  - **windrvr_gcc_v3_regparm.a** — for GCC v3.x.x compilation with the **regparm** flag
  - **windrvr_gcc_v2.a** — for GCC v2.x.x compilation; note that this file is not found in the 64-bit WinDriver installations, because 64-bit Linux architectures don't use GCC v2.
- Configuration scripts and makefile templates for creating makefiles for building and installing the WinDriver kernel driver modules.

  Files that include **.kbuild** in their names use **kbuild** for the driver compilation.

  - **configure** — a configuration script that uses the **makefile.in** template to create a **makefile** for building and installing the WinDriver driver modules, and executes the **configure.wd** and **configure.usb** scripts (see below).
  - **configure.wd** — a configuration script that uses the **makefile.wd[.kbuild].in** template to create a **makefile.wd[.kbuild]** makefile for building the **windrvr6.o/.ko** driver module.
  - **configure.usb** — a configuration script that uses the **makefile.usb[.kbuild].in** template to create a **makefile.usb[.kbuild]** makefile for building the **windrvr6_usb.o/.ko** driver module.
  - **makefile.in** — a template for the main **makefile** for building and installing the WinDriver kernel driver modules, using **makefile.wd[.kbuild]** and **makefile.usb[.kbuild]**.
  - **makefile.wd.in** and **makefile.wd.kdbuild.in** — templates for creating **makefile.wd[.kbuild]** makefiles for building and installing the **windrvr6.o/.ko** driver module.
  - **makefile.usb.in** and **makefile.usb.kdbuild.in** — templates for creating **makefile.usb[.kbuild]** makefiles for building and installing the **windrvr6_usb.o/.ko** driver module.

10.4.1.2. User-Mode Hardware-Control Application or Shared Object

Copy the user-mode hardware-control application or shared object that you created with WinDriver, to the distribution package.

If your hardware-control application/shared object uses **libwdapi1150.so** — as is the case for the WinDriver samples and generated DriverWizard projects — copy this file from the <source_dir>**/lib** directory to your distribution package.

If you are distributing a 32-bit application/shared object to a target 64-bit platform [A.2] — copy **libwdapi1150_32.so** from the **WinDriver/lib** directory to your distribution package, and rename the copy to **libwdapi1150.so**.

Since your hardware-control application/shared object does not have to be matched against the Linux kernel version number, you may distribute it as a binary object (to protect your code from unauthorized copying). If you select to distribute your driver's source code, note that under the license agreement with Jungo you may not distribute the source code of the **libwdapi1150.so** shared object, or the WinDriver license string used in your code.

10.4.2. Building and Installing the WinDriver Driver Modules on the Target

From the distribution package subdirectory containing the **configure** script and related build and installation files — normally the **redist** subdirectory [10.4.2] — perform the following steps to build and install the driver modules on the target:

1. Generate the required makefiles:
   ```
   $ ./configure
   ```

   - The configuration script creates a makefile based on the running kernel. You may select to use another installed kernel source, by executing the script with the **`--with-kernel-source=<path>`** option, where **`<path>`** is the full path to the kernel source directory — e.g., **/usr/src/linux**.
   - If the Linux kernel version is 2.6.26 or higher, the configuration script generates makefiles that use **kbuild** to compile the kernel modules. You can force the use of **kbuild** on earlier versions of Linux, by executing the configuration script with the **`--enable-kbuild`** flag.

   For a full list of the configuration script options, use the **`--help`** option:
   ```
   ./configure --help
   ```

2. Build the WinDriver driver modules:
   ```
   $ make
   ```

   This will create a **LINUX.<kernel version>.<CPU>** directory, containing the newly compiled driver modules — **windrvr6.o/.ko** and **windrvr6_usb.o/.ko**.

3. Install the **windrvr6.o/.ko** and **windrvr6_usb.o/.ko** driver modules.

   The following command must be executed with root privileges.

   ```
   # make install
   ```

The installation is performed using the **setup_inst_dir** script, which copies the driver modules to the target's loadable kernel modules directory, and uses the **wdreg** script [9.3] to load the driver modules.

4. Change the user and group IDs and give read/write permissions to the device file **/dev/windrvr6**, depending on how you wish to allow users to access hardware through the device. Due to security reasons, by default the device file is created with permissions only for the root user. Change the permissions by modifying your **/etc/udev/permissions.d/50-udev.permissions** file. For example, add the following line to provide read and write permissions:
```
windrvr6:root:root:0666
```

Use the **wdreg** script to dynamically load the WinDriver driver modules on the target after each boot [9.3]. To automate this, copy **wdreg** to the target machine, and add the following line to the target's Linux boot file (for example, **/etc/rc.local**):
```
<path to wdreg> windrvr6
```

10.4.3. Installing the User-Mode Hardware-Control Application or Shared Object

If your user-mode hardware-control application or shared object uses **libwdapi1150.so** [10.4.1.2], copy **libwdapi1150.so** from the distribution package to the target's library directory:

- **/usr/lib** — when distributing a 32-bit application/shared object to a 32-bit or 64-bit target
- **/usr/lib64** — when distributing a 64-bit application/shared object to a 64-bit target

If you decided to distribute the source code of the application/shared object [10.4.1.2], copy the source code to the target as well.

Remember that you may *not* distribute the source code of the **libwdapi1150.so** shared object or your WinDriver license string as part of the source code distribution [10.4.1.2].

Chapter 11
Driver Installation — Advanced Issues

11.1. Windows INF Files

Device information (INF) files are text files that provide information used by the Windows Plug-and-Play mechanism to install software that supports a given hardware device. INF files are required for hardware that identifies itself, such as USB and PCI. An INF file includes all necessary information about a device and the files to be installed. When hardware manufacturers introduce new products, they must create INF files to explicitly define the resources and files required for each class of device.

In some cases, the INF file for your specific device is supplied by the operating system. In other cases, you will need to create an INF file for your device. WinDriver's DriverWizard can generate a specific INF file for your device. The INF file is used to notify the operating system that WinDriver now handles the selected device.

For USB devices, you will not be able to access the device with WinDriver (either from DriverWizard or from the code) without first registering the device to work with **windrvr6.sys**. This is done by installing an INF file for the device. DriverWizard will offer to automatically generate the INF file for your device.

You can use DriverWizard to generate the INF file on the development machine — as explained in Section 5.2 of the manual — and then install the INF file on any machine to which you distribute the driver, as explained in the following sections.

11.1.1. Why Should I Create an INF File?

- To bind the WinDriver kernel module to a specific USB device.
- To override the existing driver (if any).
- To enable WinDriver applications and DriverWizard to access a USB device.

11.1.2. How Do I Install an INF File When No Driver Exists?

You must have administrative privileges in order to install an INF file.

You can use the **wdreg** utility with the **`install`** command to automatically install the INF file:
`wdreg -inf <path to the INF file> install`
(For more information, refer to Section 9.2.1 of the manual.)

On the development PC, you can have the INF file automatically installed when selecting to generate the INF file with DriverWizard, by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation window (refer to Section 5.2).

It is also possible to install the INF file manually, using either of the following methods:

- Windows **Found New Hardware Wizard**: This wizard is activated when the device is plugged in or, if the device was already connected, when scanning for hardware changes from the Device Manager.
- Windows **Add/Remove Hardware Wizard**: Right-click the mouse on **My Computer**, select **Properties**, choose the **Hardware** tab and click on **Hardware Wizard...**.
- Windows **Upgrade Device Driver Wizard**: Locate the device in the **Device Manager** devices list and select the **Update Driver...** option from the right-click mouse menu or from the Device Manager's **Action** menu.

In all the manual installation methods above you will need to point Windows to the location of the relevant INF file during the installation.
We recommend using the **wdreg** utility to install the INF file automatically, instead of installing it manually.

If the installation fails with an **`ERROR_FILE_NOT_FOUND`** error, inspect the Windows registry to see if the **`RunOnce`** key exists in **`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`**. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the **`RunOnce`** key is missing, create it; then try installing the INF file again.

11.1.3. How Do I Replace an Existing Driver Using the INF File?

You must have administrative privileges in order to replace a driver.

1. Install your INF file.

You can use the **wdreg** utility with the **`install`** command to automatically install the INF file:

```
wdreg -inf <path to INF file> install
```

(For more information, refer to Section 9.2.1 of the manual.)

On the development PC, you can have the INF file automatically installed when selecting to generate the INF file with DriverWizard, by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation window (refer to Section 5.2).

It is also possible to install the INF file manually, using either of the following methods:

- Windows **Found New Hardware Wizard**: This wizard is activated when the device is plugged in or, if the device was already connected, when scanning for hardware changes from the Device Manager.
- Windows **Add/Remove Hardware Wizard**: Right-click on **My Computer**, select **Properties**, choose the **Hardware** tab and click on **Hardware Wizard...**.
- Windows **Upgrade Device Driver Wizard**: Locate the device in the **Device Manager** devices list and select the **Update Driver...** option from the right-click mouse menu or from the Device Manager's **Action** menu.

In the manual installation methods above you will need to point Windows to the location of the relevant INF file during the installation. If the installation wizard offers to install an INF file other than the one you have generated, select **Install one of the other drivers** and choose your specific INF file from the list.

We recommend using the **wdreg** utility to install the INF file automatically, instead of installing it manually.

If the installation fails with an **`ERROR_FILE_NOT_FOUND`** error, inspect the Windows registry to see if the **`RunOnce`** key exists in **`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`**. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the **`RunOnce`** key is missing, create it; then try installing the INF file again.

11.2. Renaming the WinDriver Kernel Driver

The WinDriver APIs are implemented within the WinDriver kernel driver module (**windrvr6.sys/.dll/.o/.ko** — depending on the OS), which provides the main driver functionality and enables you to code your specific driver logic from the user mode [1.5].

On Windows and Linux you can change the name of the WinDriver kernel module to your preferred driver name, and then distribute the renamed driver instead of default kernel module — **windrvr6.sys/.o/.ko**. The following sections explain how to rename the driver for each of the supported operating systems.

A renamed WinDriver kernel driver can be installed on the same machine as the original kernel module.
You can also install multiple renamed WinDriver drivers on the same machine, simultaneously.

Try to give your driver a unique name in order to avoid a potenial conflict with other drivers on the target machine on which your driver will be installed.

11.2.1. Windows Driver Renaming

DriverWizard automates most of the work of renaming the Windows WinDriver kernel driver — **windrvr6.sys**.

- When renaming the driver, the CPU architecture (32-/64-bit) of the development platform and its WinDriver installation, should match the target platform.
- Renaming the signed **windrvr6.sys** driver nullifies its signature. In such cases you can select either to sign your new driver, or to distribute an unsigned driver. For more information on driver signing and certification, refer to Section 11.3. For guidelines for signing and certifying your renamed driver, refer to Section 11.3.2.

References to **xxx** in this section should be replaced with the name of your generated DriverWizard driver project.

To rename your Windows WinDriver kernel driver, follow these steps:

1. Use the DriverWizard utility to generate driver code for your hardware on Windows (refer to Section 5.2, Step 7), using your preferred driver name (**xxx**) as the name of the generated driver project. The generated project directory (**xxx**) will include an **xxx_installation** directory with the following files and directories:

   - **redist** directory:

     - **xxx.sys** — Your new driver, which is actually a renamed copy of the **windrvr6.sys** driver. Note: The properties of the generated driver file (such as the file's version, company name, etc.) are identical to the properties of the original **windrvr6.sys** driver. You can rebuild the driver with new properties using the files from the generated **xxx_installation sys** directory, as explained below.

     - **xxx_driver.inf** — A modified version of the **windrvr6.inf** file, which will be used to install your new **xxx.sys** driver.
       You can make additional modifications to this file, if you wish — namely, changing the string definitions and/or comments in the file.

     - **xxx_device.inf** — A modified version of the standard generated DriverWizard INF file for your device, which registers your device with your driver (**xxx.sys**).
       You can make additional modifications to this file, if you wish, such as changing the manufacturer or driver provider strings.

- **wdapi1150.dll** — A copy of the WinDriver-API DLL. The DLL is copied here in order to simplify the driver distribution, allowing you to use the generated **xxx\redist** directory as the main installation directory for your driver, instead of the original **WinDriver\redist** directory.

- **wdreg.exe**, **wdreg_gui.exe**, and **difxapi.dll** — Copies of the CUI and GUI versions of the **wdreg** WinDriver driver installation utility, and the Driver Install Frameworks API (**DIFxAPI**) DLL required by this utility [9.2.1], (respectively). These files are copied from the **WinDriver\util** directory, to simplify the installation of the renamed driver.

- **xxx_install.bat** — An installation script that executes the **wdreg** commands for installing the **xxx_driver.inf** and **xxx_device.inf** files. This script is designed to simplify the installation of the renamed **xxx_driver.sys** driver, and the registration of your device with this driver.

- **sys** directory: This directory contains files for advanced users, who wish to change the properties of their driver file. Note: Changing the file's properties requires rebuilding of the driver module using the Windows Driver Kit (**WDK**).
  To modify the properties of your **xxx.sys** driver file:

  1. Verify that the WDK is installed on your development PC, or elsewhere on its network, and set the **BASEDIR** environment variable to point to the WDK installation directory.

  2. Modify the **xxx.rc** resources file in the generated **sys** directory in order to set different driver file properties.

  3. Rebuild the driver by running the following command:
     `ddk_make <OS> <build mode (free/checked)>`
     For example, to build a release version of the driver for Windows XP:
     `ddk_make winxp free`

     - The **ddk_make.bat** utility is provided under the **WinDriver\util** directory, and should be automatically identified by Windows when running the build command. Run **`ddk_make.bat`** with no parameters to view the available options for this utility.

     - The selected build OS must match the CPU architecture of your WinDriver installation. For example, you cannot select the 64-bit `win7_x64` OS flag when using a 32-bit WinDriver installation.

  After rebuilding the **xxx.sys** driver, copy the new driver file to the generated **xxx_installation\redist** directory.

2. Verify that your user-mode application calls the `WD_DriverName()` function [B.1] with your new driver name before calling any other WinDriver function.
   Note that the sample and generated DriverWizard WinDriver applications already include a call to this function, but with the default driver name (**windrvr6**), so all you need to do is replace the driver name that is passed to the function in the code with your new driver name.

3. Verify that your user-mode driver project is built with the `WD_DRIVER_NAME_CHANGE` preprocessor flag (e.g., `-DWD_DRIVER_NAME_CHANGE`).
   Note: The sample and generated DriverWizard WinDriver kernel projects/makefiles already set this preprocessor flag by default.

4. Install your new driver by following the instructions in Section 10.2 of the manual, using the modified files from the generated **xxx_installation** directory instead of the installation files from the original WinDriver distribution. Note that you can use the generated **xxx_install.bat** installation script (see Step 1) to simplify the installation.

11.2.2. Linux Driver Renaming

DriverWizard automates most of the work of renaming the Linux WinDriver kernel driver — **windrvr6.o/.ko**.

When renaming **windrvr6.o/.ko**, the **windrvr6_usb.o/.ko** WinDriver USB Linux GPL driver is automatically renamed to **<new driver name>_usb.o/.ko**.

References to **xxx** in this section should be replaced with the name of your generated DriverWizard driver project.

To rename your Linux WinDriver kernel driver, follow these steps:

1. Use the DriverWizard utility to generate driver code for your hardware on Linux (refer to Section 5.2, Step 7), using your preferred driver name (**xxx**) as the name of the generated driver project. The generated project directory (**xxx**) will include an **xxx_installation** directory with the following files and directories:

   - **redist** directory: This directory contains copies of the files from the original **WinDriver/redist** installation directory, but with the required modifications for building your **xxx.o/.ko** driver instead of **windrvr6.o/.ko**.

   - **lib** and **include** directories: Copies of the library and include directories from the original WinDriver distribution. These copies are created since the supported Linux WinDriver kernel driver build method relies on the existence of these directories directly under the same parent directory as the **redist** directory.

2. Verify that your user-mode application calls the `WD_DriverName()` function [B.1] with your new driver name before calling any other WinDriver function.
   Note that the sample and generated DriverWizard WinDriver applications already include a call to this function, but with the default driver name (**windrvr6**), so all you need to do is replace the driver name that is passed to the function in the code with your new driver name.

3. Verify that your user-mode driver project is built with the `WD_DRIVER_NAME_CHANGE` preprocessor flag (`-DWD_DRIVER_NAME_CHANGE`).
   Note: The sample and generated DriverWizard WinDriver kernel projects/makefiles already set this preprocessor flag by default.

4. Install your new driver by following the instructions in Section 10.4 of the manual, using the modified files from the generated **xxx_installation** directory instead of the installation files from the original WinDriver distribution.
   As part of the installation, build your new kernel driver module(s) by following the instructions in Section 10.4, using the files from your new installation directory.

11.3. Windows Digital Driver Signing and Certification

11.3.1. Overview

Before distributing your driver, you may digitally sign it using Microsoft's Authenticode mechanism, and/or certify it by submitting it to Microsoft's Windows Certification Program. Some Windows operating systems, such as Windows XP, do not require installed drivers to be digitally signed or certified. There are, however, advantages to getting your driver digitally signed or fully certified, including the following:

- Driver installation on systems where installing unsigned drivers has been blocked
- Avoiding warnings during driver installation
- Full pre-installation of INF files [11.1] on Windows XP and higher

64-bit versions of Windows Vista and higher require Kernel-Mode Code Signing (KMCS) of software that loads in kernel mode. This has the following implications for WinDriver-based drivers:

- Drivers that are installed via an INF file must be distributed together with a signed catalog file (see details in Section 11.3.2).
- Drivers that are not installed using an INF file must contain an embedded driver signature.

During driver development, you can configure Windows to temporarily allow the installation of unsigned drivers.

For more information about digital driver signing and certification, refer to the following documentation in the Microsoft Development Network (MSDN) library:

- *Driver Signing Requirements for Windows*
- *Introduction to Code Signing*
- *Digital Signatures for Kernel Modules on Windows*
  This white paper contains information about kernel-mode code signing, test signing, and disabling signature enforcement during development.

Some of the documentation may still use old terminology. For example, references to the *Windows Logo Program (**WLP**)* or to the *Windows Hardware Quality Labs (**WHQL**)* should be replaced with the ***Windows Certification Program***, and references to the *Windows Quality Online Services (**Winqual**)* should be replaced with the *Windows Dev Center Hardware Dashboard Services (the **Hardware Dashboard**)*.

11.3.1.1. Authenticode Driver Signature

The Microsoft Authenticode mechanism verifies the authenticity of a driver's provider. It allows driver developers to include information about themselves and their code with their programs through the use of digital signatures, and informs users of the driver that the driver's publisher is participating in an infrastructure of trusted entities.
The Authenticode signature does not, however, guarantee the code's safety or functionality.

The **WinDriver\redist\windrvr6.sys** driver has an Authenticode digital signature.

11.3.1.2. Windows Certification Program

Microsoft's Windows Certification Program (previously known as the Windows Logo Program (WLP)), lays out procedures for submitting hardware and software modules, including drivers, for Microsoft quality assurance tests. Passing the tests qualifies the hardware/software for Microsoft certification, which verifies both the driver provider's authenticity and the driver's safety and functionality.

To digitally sign and certify a device driver, a Windows Hardware Certification Kit (**HCK**) package, which includes the driver and the related hardware, should be submitted to the Windows Certification Program for testing, using the Windows Dev Center Hardware Dashboard Services (the **Hardware Dashboard**).

Jungo's professional services unit provides a complete Windows driver pre-certification service for Jungo-based drivers. Professional engineers efficiently perform all the tests required by the Windows Certification Program, relieving customers of the expense and stress of in-house testing. Jungo prepares an HCK submission package containing the test results, and delivers the package to the customer, ready for submission to Microsoft.
For more information, refer to
http://www.jungo.com/st/products/windriver/windriver_whql_certification/.

For detailed information regarding Microsoft's Windows Certification Program and the certification process, refer to the MSDN *Windows Hardware Certification* page — http://msdn.microsoft.com/library/windows/hardware/gg463010.aspx — and to the documentation referenced from that page, including the MSDN *Windows Dev Center — Hardware Dashboard Services* page — http://msdn.microsoft.com/library/windows/hardware/gg463091.

11.3.2. Driver Signing and Certification of WinDriver-Based Drivers

As indicated above [11.3.1.1], The **WinDriver\redist\windrvr6.sys** driver has an Authenticode signature. Since WinDriver's kernel module (**windrvr6.sys**) is a generic driver, which can be used as a driver for different types of hardware devices, it cannot be submitted to Microsoft's Windows Certification Program as a standalone driver. However, once you have used WinDriver to develop a Windows driver for your selected hardware, you can submit both the hardware and driver for Microsoft certification, as explained below.

The driver certification and signature procedures — either via Authenticode or the Windows Certification Program — require the creation of a catalog file for the driver. This file is a sort of hash, which describes other files. The signed **windrvr6.sys** driver is provided with a matching catalog file — **WinDriver\redist\wd1150.cat**. This file is assigned to the `CatalogFile` entry in the **windrvr6.inf** file (provided as well in the **redist** directory). This entry is used to inform Windows of the driver's signature and the relevant catalog file during the driver's installation.

When the name, contents, or even the date of the files described in a driver's catalog file is modified, the catalog file, and consequently the driver signature associated with it, become invalid. Therefore, if you select to rename the **windrvr6.sys** driver [11.2] and/or the related **windrvr6.inf** file, the **wd1150.cat** catalog file and the related driver signature will become invalid.

In addition, when using WinDriver to develop a driver for your Plug-and-Play device, you normally also create a device-specific INF file that registers your device to work with the **windrvr6.sys** driver module (or a renamed version of this driver). Since this INF file is created at your site, for your specific hardware, it is not referenced from the **wd1150.cat** catalog file and cannot be signed by Jungo a priori.

When renaming **windrvr6.sys** and/or creating a device-specific INF file for your device, you have two alternative options regarding your driver's digital signing:

- Do not digitally sign your driver. If you select this option, remove or comment-out the reference to the **wd1150.cat** file from the **windrvr6.inf** file (or your renamed version of this file).

- Submit your driver to the Windows Certification Program, or have it Authenticode signed. Note that while renaming **WinDriver\redist\windrvr6.sys** nullifies the driver's digital signature, the driver is still compliant with the certification requirements of the Windows Certification Program.

To digitally sign/certify your driver, follow these steps:

- Create a new catalog file for your driver, as explained in the Windows Certification Program documentation. The new file should reference both **windrvr6.sys** (or your renamed driver) and any INF files used in your driver's installation.
- Assign the name of your new catalog file to the `CatalogFile` entry in your driver's INF file(s). (You can either change the `CatalogFile` entry in the **windrvr6.inf** file to refer to your new catalog file, and add a similar entry in your device-specific INF file; or incorporate both **windrvr6.inf** and your device INF file into a single INF file that contains such a `CatalogFile` entry).
- Submit your driver to Microsoft's Windows Certification Program or for an Authenticode signature. If you wish to submit your driver to the Windows Certification Program, refer to the additional guidelines in Section 11.3.2.1.

  Note that many WinDriver customers have already successfully digitally signed and certified their WinDriver-based drivers.

11.3.2.1. HCK Test Notes

As indicated in Microsoft's documentation, before submitting the driver for testing and certification you need to download the Windows Hardware Certification Kit (**HCK**), and run the relevant tests for your hardware/software. After you have verified that you can successfully pass the HCK tests, create the required logs package and proceed according to Microsoft's documentation. For more information, refer to the MSDN *Windows Hardware Certification Kit (HCK)* page — http://msdn.microsoft.com/library/windows/hardware/hh833788.

When running the HCK tests, note the following:

- The Driver Verifier test is applied to all unsigned drivers found on the test machine. It is therefore important to try and minimize the number of unsigned drivers installed on the test machine (apart from the test driver — **windrvr6.sys**).
- The USB Selective Suspend test requires that the depth of the under-test USB device in the USB devices tree is at least one external hub and no more than two external hubs deep.
- The ACPI Stress test requires that the ACPI settings in the BIOS support the S3 power state.
- Before submitting the file for certification you need to create a new catalog file, which lists your driver and specific INF file(s), and refer to this catalog file from your INF file(s), as explained above [11.3.2].

11.4. Windows XP Embedded WinDriver Component

When creating a Windows XP Embedded image using the Target Designer tool from Microsoft's Windows Embedded Studio, you can select the components that you wish to add to your image. The added components will be installed automatically during the first boot on the Windows XP Embedded target on which the image is loaded.

To automatically install the required WinDriver files — such as the **windrvr6.inf** file and the WinDriver kernel driver that it installs (**windrvr6.sys**), your device INF file , and the WinDriver-API DLL (**wdapi1150.dll**) — on Windows XP Embedded platforms, you can create a relevant WinDriver component and add it to your Windows XP Embedded image.
WinDriver simplifies this task for you by providing you with a ready-made component: **WinDriver\redist\xp_embedded\wd_component\windriver.sld**.
To use the provided component, follow the steps below.

The provided **windriver.sld** component relies on the existence of a **wd_files** directory in the same directory that holds the component. Therefore, do not rename the provided **WinDriver\redist\xp_embedded\wd_component\wd_files** directory or modify its contents, unless instructed to so in the following guidelines.

1. **Modify the dev.inf file:**
   The **windriver.sld** component depends on the existence of a **dev.inf** file in the **wd_files** directory. The WinDriver installation on your development Windows platform contains a generic **WinDriver\redist\xp_embedded\wd_component\wd_files\dev.inf** file. Use either of the following methods to modify this file to suit your device:

   - Modify the generic **dev.inf** file to describe your device. At the very least, you must modify the template `[DeviceList]` entry and insert your device's hardware type and vendor and product IDs. For example, for a device with vendor ID 0x1234 and product ID 0x5678:
     ```
     "my_dev_usb"=Install, USB\VID_1234\&PID_5678
     ```

   OR:

   - Create an INF file for your device using DriverWizard (refer to Section 5.2, Step 3) and name it **dev.inf**. Then copy your **dev.inf** device INF file to the **WinDriver\redist\xp_embedded\wd_component\wd_files** directory.

2. **Add the WinDriver component to the Windows Embedded Component Database:**

   1. Open the Windows Embedded Component Database Manager (DBMgr).

   2. Click **Import**.

   3. Select the WinDriver component — **WinDriver\redist\xp_embedded\wd_component\windriver.sld** — as the SLD file and click **Import**.

3. **Add the WinDriver component to your Windows XP Embedded image:**

    1. Open your project in the Target Designer.

    2. Double-click the WinDriver component to add it to your project.
       Note: If you already have an earlier version of the WinDriver component in your project's components list, right-click this component and select **Upgrade**.

    3. Run a dependency check and build your image.

After following these steps, WinDriver will automatically be installed during the first boot on the target Windows XP Embedded platform on which your image is loaded.

If you have selected to rename the WinDriver kernel module [11.2], you will not be able to use the provided **windriver.sld** component. You can build your own component for the renamed driver, or use the **wdreg** utility to install the driver on the target Windows XP Embedded platform, as explained in the manual.

Appendix A
64-Bit Operating Systems Support

A.1. Supported 64-Bit Architectures

WinDriver supports the following 64-bit platforms:

- Linux AMD64 or Intel EM64T (**x86_64**)
  For a full list of the Linux platforms supported by WinDriver, refer to Section 4.1.3.

- Windows AMD64 or Intel EM64T (**x64**).
  For a full list of the Windows platforms supported by WinDriver, refer to Section 4.1.1.

A.2. Support for 32-Bit Applications on 64-Bit Windows and Linux Platforms

By default, applications created using the 64-bit versions of WinDriver are 64-bit applications. Such applications are more efficient than 32-bit applications. However, you can also use the 64-bit WinDriver versions to create 32-bit applications that will run on the supported Windows and Linux 64-bit platforms [A.1].

In the following documentation, <**WD64**> signifies the path to a 64-bit WinDriver installation directory for your target operating system, and <**WD32**> signifies the path to a 32-bit WinDriver installation directory for the same operating system.

To create a 32-bit application for 64-bit Windows or Linux platforms, using the 64-bit version of WinDriver, do the following:

1. Create a WinDriver application, as outlined in this manual (e.g., by generating code with DriverWizard, or using one of the WinDriver samples).

2. Build the application with an appropriate 32-bit compiler for your target OS, using the following configuration:

   - Add a **`KERNEL_64BIT`** preprocessor definition to your project or makefile.

     In the makefiles, the definition is added using the `-D` flag: `-DKERNEL_64BIT`.

     The sample and wizard-generated Linux and Windows GCC makefiles and the Windows MS Visual Studio projects, in the 64-bit WinDriver toolkit, already add this definition.

- Link the application with the specific version of the WinDriver-API library/ shared object for 32-bit applications executed on 64-bit platforms — **<WD64>\lib\amd64\x86\wdapi1150.lib** on Windows / **<WD64>/lib/libwdapi1150_32.so** on Linux.

  The sample and wizard-generated project and make files for 32-bit applications in the 64-bit WinDriver toolkit already link to the correct library:
  On Windows, the MS Visual Studio project files and Windows GCC makefiles are defined to link with **<WD64>\lib\amd64\x86\wdapi1150.lib**.
  On Linux, the installation of the 64-bit WinDriver toolkit on the development machine creates a **libwdapi1150.so** symbolic link in the **/usr/lib** directory — which links to **<WD64>/lib/libwdapi1150_32.so** — and in the **/usr/lib64** directory — which links to **<WD64>/lib/libwdapi1150.so** (the 64-bit version of this shared object).
  The sample and wizard-generated WinDriver makefiles rely on these symbolic links to link with the appropriate shared object, depending on whether the code is compiled using a 32-bit or 64-bit compiler.

- When distributing your application to target 64-bit platforms, you need to provide with it the WinDriver-API DLL/shared object for 32-bit applications executed on 64-bit platforms — **<WD64>\redist\wdapi1150_32.dll** on Windows / **<WD64>/lib/libwdapi1150_32.so** on Linux. Before distributing this file, rename the copy of the file in your distribution package by removing the **_32** portion. The installation on the target should copy the renamed DLL/shared object to the relevant OS directory — **\%windir%\sysWOW64** on Windows or **/usr/lib** on Linux. All other distribution files are the same as for any other 64-bit WinDriver driver distribution, as detailed in Chapter 10.

- An application created using the method described in this section will *not* work on 32-bit platforms. A WinDriver application for 32-bit platforms needs to be compiled without the `KERNEL_64BIT` definition; it needs to be linked with the standard 32-bit version of the WinDriver-API library/shared object from the 32-bit WinDriver installation (**<WD32>\lib\x86\wdapi1150.lib** on Windows / **<WD32>/lib/libwdapi1150.so** on Linux); and it should be distributed with the standard 32-bit WinDriver-API DLL/shared object (**<WD32>\redist\wdapi1150.dll** on Windows / **<WD32>/lib/libwdapi1150.so** on Linux) and any other required 32-bit distribution file, as outlined in Chapter 10.

A.3. 64-Bit and 32-Bit Data Types

In general, DWORD is unsigned long. While any 32-bit compiler treats this type as 32 bits wide, 64-bit compilers treat this type differently. With Windows 64-bit compilers the size of this type is still 32 bits. However, with UNIX 64-bit compilers (e.g., GCC) the size of this type is 64 bits. In order to avoid compiler dependency issues, use the UINT32 and UINT64 cross-platform types when you want to refer to a 32-bit or 64-bit address, respectively.

Appendix B
WinDriver USB Host API Reference

This function reference is C oriented. The WinDriver C# and Visual Basic .NET APIs have been implemented as closely as possible to the C APIs, therefore .NET programmers can also use this reference to better understand the WinDriver APIs for their selected development language. For the exact API implementation and usage examples for your selected language, refer to the WinDriver .NET source code.

B.1. WD_DriverName

Purpose

Sets the name of the WinDriver kernel module, which will be used by the calling application.

- The default driver name, which is used if the function is not called, is **windrvr6**.
- This function must be called once, and only once, from the beginning of your application, before calling any other WinDriver function (including `WD_Open`() / `WDU_Init`()), as demonstrated in the sample and generated DriverWizard WinDriver applications, which include a call to this function with the default driver name — **windrvr6**.
- On Windows and Linux, if you select to modify the name of the WinDriver kernel module (**windrvr6.sys/.o/.ko**), as explained in Section 11.2, you must ensure that your application calls `WD_DriverName`() with your new driver name.
- In order to use the `WD_DriverName`() function, your user-mode driver project must be built with `WD_DRIVER_NAME_CHANGE` preprocessor flag (e.g.: `-DWD_DRIVER_NAME_CHANGE` — for MS Visual Studio, Windows GCC, and GCC).
  The sample and generated DriverWizard Windows and Linux WinDriver projects/makefiles already set this preprocessor flag.

Prototype

```
const char* DLLCALLCONV WD_DriverName(const char* sName);
```

Parameters

Name	Type	Input/Output
sName	const char*	Input

Description

Name	Description
sName	The name of the WinDriver kernel module to be used by the application. NOTE: The driver name should be indicated without the driver file's extension. For example, use **windrvr6**, not **windrvr6.sys** or **windrvr6.o**.

Return Value

Returns the selected driver name on success; returns NULL on failure (e.g., if the function is called twice from the same application)long.

Remarks

The ability to rename the WinDriver kernel module is supported on Windows and Linux, as explained in Section 11.2.
On Windows CE, always call the `WD_DriverName()` function with the default WinDriver kernel module name — **windrvr6** — or refrain from calling the function altogether.

B.2. WinDriver USB (WDU) Library Overview

This section provides a general overview of WinDriver's USB Library (WDU), including

- An outline of the `WDU_xxx` API calling sequence — see Section B.2.1.
- Instructions for upgrading code developed with the previous WinDriver USB API, used in version 5.22 and earlier, to use the improved `WDU_xxx` API — see Section B.2.2.
  If you do not need to upgrade USB driver code developed with an older version of WinDriver, simply skip this section.

The WDU library's interface is found in the **WinDriver/include/wdu_lib.h** and **WinDriver/include/windrvr.h** header files, which should be included from any source file that calls the WDU API. (**wdu_lib.h** already includes **windrvr.h**).

B.2.1. Calling Sequence for WinDriver USB

The WinDriver `WDU_xxx` USB API is designed to support event-driven transfers between your user-mode USB application and USB devices. This is in contrast to earlier versions, in which USB devices were initialized and controlled using a specific sequence of function calls.

You can implement the three user callback functions specified in the next section: `WDU_ATTACH_CALLBACK` [B.3.1], `WDU_DETACH_CALLBACK` [B.3.2] and `WDU_POWER_CHANGE_CALLBACK` [B.3.3] (at the very least `WDU_ATTACH_CALLBACK`).

These functions are used to notify your application when a relevant system event occurs, such as the attaching or detaching of a USB device. For best performance, minimal processing should be done in these functions.

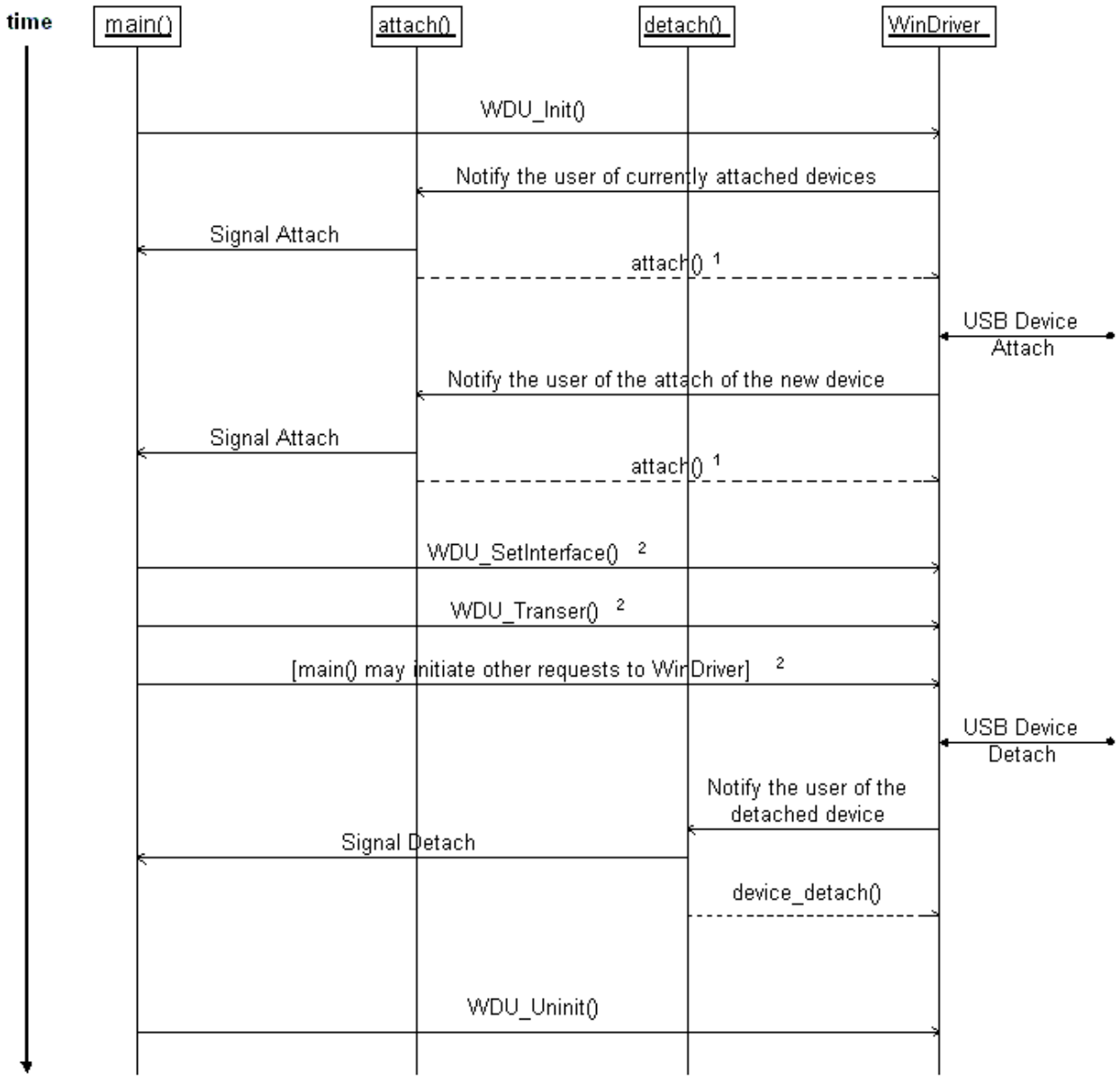
Your application calls `WDU_Init`() [B.4.1] and provides the criteria according to which the system identifies a device as relevant or irrelevant. The `WDU_Init`() function must also pass pointers to the user callback functions.

Your application then simply waits to receive a notification of an event. Upon receipt of such a notification, processing continues. Your application may make use of any functions defined in the high- or low-level APIs below. The high-level functions, provided for your convenience, make use of the low-level functions, which in turn use IOCTLs to enable communication between the WinDriver kernel module and your user-mode application.

When exiting, your application calls `WDU_Uninit`() [B.4.7] to stop listening to devices matching the given criteria and to unregister the notification callbacks for these devices.

The following figure depicts the calling sequence described above. Each vertical line represents a function or process. Each horizontal arrow represents a signal or request, drawn from the initiator to the recipient. Time progresses from top to bottom.

Figure B.1. WinDriver USB Calling Sequence



[1] If the WD_ACKNOWLEDGE flag was set in the call to WDU_Init(), the attach() callback should return TRUE to accept control of the device or FALSE otherwise.

[2] Only possible if the attach() callback returned TRUE.

The following piece of meta-code can serve as a framework for your user-mode application's code:

```
attach()
{
    ...
    if this is my device
        /*
        Set the desired alternate setting ;
        Signal main() about the attachment of this device
        */

        return TRUE;
    else
        return FALSE;
}

detach()
{
    ...
    signal main() about the detachment of this device
    ...
}

main()
{
    WDU_Init(...);

    ...
    while (...)
    {
        /* wait for new devices */

        ...

        /* issue transfers */

        ...
    }
    ...
    WDU_Uninit();
}
```

B.2.2. Upgrading from the WD_xxx USB API to the WDU_xxx API

The WinDriver `WDU_xxx` USB API, provided beginning with version 6.00, is designed to support event-driven transfers between your user-mode USB application and USB devices. This is in contrast to earlier versions, in which USB devices were initialized and controlled using a specific sequence of function calls.

As a result of this change, you will need to modify your USB applications that were designed to interface with earlier versions of WinDriver to ensure that they will work with WinDriver v6.X on all supported platforms and not only on Microsoft Windows.
You will have to reorganize your application's code so that it conforms with the framework illustrated by the piece of meta-code provided in Section B.2.1.

In addition, the functions that collectively define the USB API have been changed. The new functions, described in the next few sections, provide an improved interface between user-mode USB applications and the WinDriver kernel module. Note that the new functions receive their parameters directly, unlike the old functions, which received their parameters using a structure.

The table below lists the legacy functions in the left column and indicates in the right column which function or functions replace(s) each of the legacy functions. Use this table to quickly determine which new functions to use in your new code.

Problem	**Solution**
High Level API	
Previous Function	*New Function*
`WD_Open`() `WD_Version`() `WD_UsbScanDevice`()	`WDU_Init`() [B.4.1]
`WD_UsbDeviceRegister`()	`WDU_SetInterface`() [B.4.2]
`WD_UsbGetConfiguration`()	`WDU_GetDeviceInfo`() [B.4.5]
`WD_UsbDeviceUnregister`()	`WDU_Uninit`() [B.4.7]
Low Level API	
Previous Function	*New Function*
`WD_UsbTransfer`()	`WDU_Transfer`() [B.4.8.1] `WDU_TransferDefaultPipe`() [B.4.8.3] `WDU_TransferBulk`() [B.4.8.4] `WDU_TransferIsoch`() [B.4.8.5] `WDU_TransferInterrupt`() [B.4.8.6]
`USB_TRANSFER_HALT` option	`WDU_HaltTransfer`() [B.4.8.2]
`WD_UsbResetPipe`()	`WDU_ResetPipe`() [B.4.10]
`WD_UsbResetDevice`() `WD_UsbResetDeviceEx`()	`WDU_ResetDevice`() [B.4.11]

B.3. USB User Callback Functions

B.3.1. WDU_ATTACH_CALLBACK

Purpose

WinDriver calls this function when a new device, matching the given criteria, is attached, provided it is not yet controlled by another driver.
This callback is called once for each matching interface.

Prototype

```
typedef BOOL (DLLCALLCONV *WDU_ATTACH_CALLBACK)(
    WDU_DEVICE_HANDLE hDevice,
    WDU_DEVICE *pDeviceInfo,
    PVOID pUserData);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
pDeviceInfo	WDU_DEVICE*	Input
pUserData	PVOID	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface
pDeviceInfo	Pointer to a USB device information structure [B.5.2.3]; Valid until the end of the function
pUserData	Pointer to user-mode data for the callback, as passed to `WDU_Init`() [B.4.1] within the event table parameter (`pEventTable->pUserData`)

Return Value

If the `WD_ACKNOWLEDGE` flag was set in the call to `WDU_Init`() [B.4.1] (within the `dwOptions` parameter), the callback function should check if it wants to control the device, and if so return `TRUE` (otherwise — return `FALSE`).
If the `WD_ACKNOWLEDGE` flag was not set in the call to `WDU_Init`(), then the return value of the callback function is insignificant.

B.3.2. WDU_DETACH_CALLBACK

Purpose

WinDriver calls this function when a controlled device has been detached from the system.

Prototype

```
typedef void (DLLCALLCONV *WDU_DETACH_CALLBACK)(
    WDU_DEVICE_HANDLE hDevice,
    PVOID pUserData);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
pUserData	PVOID	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface
pUserData	Pointer to user-mode data for the callback, as passed to `WDU_Init`() [B.4.1] within the event table parameter (`pEventTable->pUserData`)

Return Value

None

B.3.3. WDU_POWER_CHANGE_CALLBACK

Purpose

WinDriver calls this function when a controlled device has changed its power settings.

Prototype

```
typedef BOOL (DLLCALLCONV *WDU_POWER_CHANGE_CALLBACK)(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPowerState,
    PVOID pUserData);
```

Parameters

Name	Type	Input/Output
dwPowerState	DWORD	Input
pUserData	PVOID	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface
dwPowerState	Number of the power state selected
pUserData	Pointer to user-mode data for the callback, as passed to `WDU_Init()` [B.4.1] within the event table parameter (`pEventTable->pUserData`)

Return Value

`TRUE/FALSE`. Currently there is no significance to the return value.

Remarks

This callback is supported only on Windows.

B.4. USB Functions

The functions described in this section are declared in the **WinDriver/include/wdu_lib.h** header file.

B.4.1. WDU_Init

Purpose

Starts listening to devices matching input criteria and registers notification callbacks for these devices.

Prototype

```
DWORD WDU_Init(
    WDU_DRIVER_HANDLE *phDriver,
    WDU_MATCH_TABLE *pMatchTables,
    DWORD dwNumMatchTables,
    WDU_EVENT_TABLE *pEventTable,
    const char *sLicense,
    DWORD dwOptions);
```

Parameters

Name	Type	Input/Output
phDriver	WDU_DRIVER_HANDLE *	Output
pMatchTables	WDU_MATCH_TABLE*	Input
dwNumMatchTables	DWORD	Input
pEventTable	WDU_EVENT_TABLE*	Input
sLicense	const char*	Input
dwOptions	DWORD	Input

Description

Name	Description
phDriver	Handle to the registration of events & criteria
pMatchTables	Array of match tables [B.5.2.1] defining the devices' criteria
dwNumMatchTables	Number of elements in pMatchTables
pEventTable	Pointer to an event table structure [B.5.2.2], which holds the addresses of the user-mode device status change notification callback functions [B.3] and the data to pass to the callbacks
sLicense	WinDriver's license string
dwOptions	Can be zero or : • **`WD_ACKNOWLEDGE`** — the user can seize control over the device when returning value in `WDU_ATTACH_CALLBACK` [B.3.1]

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.2. WDU_SetInterface

Purpose

Sets the alternate setting for the specified interface.

Prototype

```
DWORD WDU_SetInterface(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwInterfaceNum,
    DWORD dwAlternateSetting);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
dwInterfaceNum	DWORD	Input
dwAlternateSetting	DWORD	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface
dwInterfaceNum	The interface's number
dwAlternateSetting	The desired alternate setting value

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

On **Windows CE** — as opposed to Windows — `WDU_SetInterface`() attempts to open all the pipes of the specified alternate setting, even if not all pipes are currently required. The reason for this is that Windows CE limits the total number of pipes that can be opened simultaneously on a device, to 16 (see http://msdn.microsoft.com/en-us/library/ms919318.aspx). By opening all the pipes, the driver ensures that the pipes will be available for use, when needed.
The pipes are opened using the Windows CE USB host controller driver's `LPOPEN_PIPE` callback. On some Windows CE devices, the call to this callback fails, causing `WDU_SetInterface`() to fail as well. To resolve such problems, upgrade the device's USB host controller driver.

B.4.3. WDU_GetDeviceAddr

Purpose

Gets the USB address for a given device.

Prototype

```
DWORD WDU_GetDeviceAddr(
    WDU_DEVICE_HANDLE hDevice,
    ULONG *pAddress);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
pAddress	ULONG	Output

Description

Name	Description
hDevice	A unique identifier for a device/interface
pAddress	A pointer to the address number returned by the function

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

This function is supported only on Windows.

B.4.4. WDU_GetDeviceRegistryProperty

Purpose

Gets the specified registry property of a given USB device.

Prototype

```
DWORD DLLCALLCONV WDU_GetDeviceRegistryProperty(
    WDU_DEVICE_HANDLE hDevice,
    PVOID pBuffer,
    PDWORD pdwSize,
    WD_DEVICE_REGISTRY_PROPERTY property);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
pBuffer	PVOID	Output
pdwSize	PDWORD	Input/Output
property	WD_DEVICE_REGISTRY_PROPERTY	Input

Description

Name	Description
hDevice	A unique identifier of the device/interface
pBuffer	Pointer to a user allocated buffer to be filled with the requested registry property. The function will fill the buffer only if the buffer size, as indicated in the input value of the **`pdwSize`** parameter, is sufficient — i.e., >= the property's size, as returned via **`pdwSize`**. **`pBuffer`** can be set to `NULL` when using the function only to retrieve the size of the registry property (see **`pdwSize`**).
pdwSize	As input, points to a value indicating the size of the user-supplied buffer (**`pBuffer`**); if **`pBuffer`** is set to `NULL`, the input value of this parameter is ignored. As output, points to a value indicating the required buffer size for storing the registry property.
property	The ID of the registry property to be retrieved — see the description of the `WD_DEVICE_REGISTRY_PROPERTY` enumeration [B.5.1]. Note: String registry properties are in WCHAR format.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

- When the size of the provided user buffer (**`pBuffer`**) — **`*pdwSize`** (input) — is not sufficient to hold the requested registry property, the function returns `WD_INVALID_PARAMETER`.

- This function is supported only on Windows.

B.4.5. WDU_GetDeviceInfo

Purpose

Gets configuration information from a device, including all the device descriptors.

NOTE: The caller to this function is responsible for calling `WDU_PutDeviceInfo`() [B.4.6] in order to free the `*ppDeviceInfo` pointer returned by the function.

Prototype

```
DWORD WDU_GetDeviceInfo(
    WDU_DEVICE_HANDLE hDevice,
    WDU_DEVICE **ppDeviceInfo);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
ppDeviceInfo	WDU_DEVICE**	Output

Description

Name	Description
hDevice	A unique identifier for a device/interface
ppDeviceInfo	Pointer to pointer to a USB device information structure [B.5.2.3]

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.6. WDU_PutDeviceInfo

Purpose

Receives a device information pointer, allocated with a previous `WDU_GetDeviceInfo()` [B.4.5] call, in order to perform the necessary cleanup.

Prototype

```
void WDU_PutDeviceInfo(WDU_DEVICE *pDeviceInfo);
```

Parameters

Name	Type	Input/Output
pDeviceInfo	WDU_DEVICE*	Input

Description

Name	Description
pDeviceInfo	Pointer to a USB device information structure [B.5.2.3], as returned by a previous call to `WDU_GetDeviceInfo()` [B.4.5]

Return Value

None

B.4.7. WDU_Uninit

Purpose

Stops listening to devices matching a given criteria and unregisters the notification callbacks for these devices.

Prototype

```
void WDU_Uninit(WDU_DRIVER_HANDLE hDriver);
```

Parameters

Name	Type	Input/Output
hDriver	WDU_DRIVER_HANDLE	Input

Description

Name	Description
hDriver	Handle to the registration received from `WDU_Init()` [B.4.1]

Return Value

None

B.4.8. Single-Blocking Transfer Functions

This section describes WinDriver's single-blocking data transfer functions.
For more information, refer to Section 8.3.2 of the manual.

B.4.8.1. WDU_Transfer

Purpose

Transfers data to or from a device.

Prototype

```
DWORD WDU_Transfer(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    PBYTE pSetupPacket,
    DWORD dwTimeout);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
dwPipeNum	DWORD	Input
fRead	DWORD	Input
dwOptions	DWORD	Input
pBuffer	PVOID	Input
dwBufferSize	DWORD	Input
pdwBytesTransferred	PDWORD	Output
pSetupPacket	PBYTE	Input
dwTimeout	DWORD	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface received from `WDU_Init()` [B.4.1]
dwPipeNum	The number of the pipe through which the data is transferred
fRead	`TRUE` for read, `FALSE` for write
dwOptions	A bit-mask of USB transfer options, which can consist of a combination of any of the following flags: • **`USB_ISOCH_NOASAP`** — Instructs the lower USB stack driver (**usbd.sys**) to use a preset frame number (instead of the next available frame) for an isochronous data transfer. It is recommended that you use this flag for isochronous write (OUT) transfers, and if you notice unused frames during transfers on low-speed or full-speed USB 1.1 devices. This flag is available only for Windows. • **`USB_ISOCH_FULL_PACKETS_ONLY`** — Prevents transfers of less than the packet size on isochronous pipes. • **`USB_BULK_INT_URB_SIZE_OVERRIDE_128K`** — Limits the size of the USB Request Block (URB) to 128KB. This flag is available only for Windows. • **`USB_ISOCH_RESET`** — Resets the isochronous pipe before the data transfer. It also resets the pipe after minor errors, consequently allowing to transfer to continue.
pBuffer	Address of the data buffer
dwBufferSize	Number of bytes to transfer. The buffer size is not limited to the device's maximum packet size; therefore, you can use larger buffers by setting the buffer size to a multiple of the maximum packet size. Use large buffers to reduce the number of context switches and thereby improve performance.
pdwBytesTransferred	Number of bytes actually transferred
pSetupPacket	An 8-byte packet to transfer to control pipes
dwTimeout	Maximum time, in milliseconds (ms), to complete a transfer. A value of zero indicates no timeout (infinite wait).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

The resolution of the timeout (the **`dwTimeout`** parameter) is according to the operating system scheduler's time slot. For example, in Windows the timeout's resolution is 10 milliseconds (ms).

B.4.8.2. WDU_HaltTransfer

Purpose

Halts the transfer on the specified pipe (only one simultaneous transfer per pipe is allowed by WinDriver).

Prototype

```
DWORD WDU_HaltTransfer(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
dwPipeNum	DWORD	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface
dwPipeNum	The number of the pipe

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.8.3. WDU_TransferDefaultPipe

Purpose

Transfers data to or from a device through the default control pipe (pipe 0).

Prototype

```
DWORD WDU_TransferDefaultPipe(
    WDU_DEVICE_HANDLE hDevice,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    PBYTE pSetupPacket,
    DWORD dwTimeout);
```

Refer to the `WDU_Transfer()` parameters documentation [B.4.8.1], except for `dwPipeNum` (N/A).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.8.4. WDU_TransferBulk

Purpose

Performs bulk data transfer to or from a device.

Prototype

```
DWORD WDU_TransferBulk(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    DWORD dwTimeout);
```

Refer to the `WDU_Transfer()` parameters documentation [B.4.8.1], except for `pSetupPacket` (N/A).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.8.5. WDU_TransferIsoch

Purpose

Performs isochronous data transfer to or from a device.

Prototype

```
DWORD WDU_TransferIsoch(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    DWORD dwTimeout);
```

Refer to the `WDU_Transfer()` parameters documentation [B.4.8.1], except for `pSetupPacket` (N/A).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.8.6. WDU_TransferInterrupt

Purpose

Performs interrupt data transfer to or from a device.

Prototype

```
DWORD WDU_TransferInterrupt(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    DWORD dwTimeout);
```

Refer to the `WDU_Transfer()` parameters documentation [B.4.8.1], except for `pSetupPacket` (N/A).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.9. Streaming Data Transfer Functions

This section describes WinDriver's streaming data transfer functions.
For a detailed explanation regarding stream transfers and their implementation with Windriver, refer to Section 8.3.3 of the manual.

The streaming APIs are currently supported on Windows and Windows CE.

B.4.9.1. WDU_StreamOpen

Purpose

Opens a new stream for the specified pipe.
A stream can be associated with any pipe except for the control pipe (pipe 0). The stream's data transfer direction — read/write — is derived from the direction of its pipe.

Prototype

```
DWORD DLLCALLCONV WDU_StreamOpen(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD dwBufferSize,
    DWORD dwRxSize,
    BOOL  fBlocking,
    DWORD dwOptions,
    DWORD dwRxTxTimeout,
    WDU_STREAM_HANDLE *phStream);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
dwPipeNum	DWORD	Input
dwBufferSize	DWORD	Input
dwRxSize	DWORD	Input
fBlocking	BOOL	Input
dwOptions	DWORD	Input
dwRxTxTimeout	DWORD	Input
phStream	WDU_STREAM_HANDLE*	Output

Description

Name	Description
hDevice	A unique identifier for the device/interface
dwPipeNum	The number of the pipe for which to open the stream
dwBufferSize	The size, in bytes, of the stream's data buffer
dwRxSize	The size, in bytes, of the data blocks that the stream reads from the device. This parameter is relevant only for read streams, and must not exceed the value of the **dwBufferSize** parameter. Note: When setting the **USB_STREAM_MAX_TRANSFER_SIZE_OVERWRITE dwOptions** flag, this is also the maximum transfer size.
fBlocking	• TRUE for a blocking stream, which performs blocked I/O; • FALSE for a non-blocking stream, which performs non-blocking I/O. For additional information, refer to Section 8.3.3.1.
dwOptions	A bit-mask of USB transfer options, which can consist of a combination of any of the following flags: • **USB_ISOCH_NOASAP** — Instructs the lower USB stack driver (**usbd.sys**) to use a preset frame number (instead of the next available frame) for an isochronous data transfer. It is recommended that you use this flag for isochronous write (OUT) transfers, and if you notice unused frames during transfers on low-speed or full-speed USB 1.1 devices. This flag is available only for Windows. • **USB_ISOCH_FULL_PACKETS_ONLY** — Prevents transfers of less than the packet size on isochronous pipes. • **USB_BULK_INT_URB_SIZE_OVERRIDE_128K** — Limits the size of the USB Request Block (URB) to 128KB. This flag is available only for Windows. • **USB_STREAM_OVERWRITE_BUFFER_WHEN_FULL** — When there is not enough free space in a read stream's data buffer to complete the transfer, overwrite old data in the buffer. This flag is applicable only to read streams. • **USB_STREAM_MAX_TRANSFER_SIZE_OVERRIDE** — Overrides the default maximum transfer size with the **dwRxSize** transfer size, on Windows CE. Note that setting a large dwRxSize value when using this flag, may cause the transfers to fail due to host controller limitations. This flag is applicable only to read streams on Windows CE.
dwRxTxTimeout	Maximum time, in milliseconds (ms), for the completion of a data transfer between the stream and the device. A value of zero indicates no timeout (infinite wait).
phStream	Pointer to a unique identifier for the stream, to be returned by the function and passed to the other WDU_StreamXXX() functions

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.9.2. WDU_StreamStart

Purpose

Starts a stream, i.e., starts transfers between the stream and the device.
Data will be transferred according to the stream's direction — read/write.

Prototype

```
DWORD DLLCALLCONV WDU_StreamStart(
    WDU_STREAM_HANDLE hStream);
```

Parameters

Name	Type	Input/Output
hStream	WDU_STREAM_HANDLE	Input

Description

Name	Description
hStream	A unique identifier for the stream, as returned by `WDU_StreamOpen()`

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.9.3. WDU_StreamRead

Purpose

Reads data from a read stream to the application.

For a blocking stream (`fBlocking=TRUE` — see `WDU_StreamOpen`()), the call to this function is blocked until the specified amount of data (**`bytes`**) is read, or until the stream's attempt to read from the device times out (i.e., the timeout period for transfers between the stream and the device, as set in the `dwRxTxTimeout` `WDU_StreamOpen`() parameter [B.4.9.1], expires).

For a non-blocking stream (`fBlocking=FALSE`), the function transfers to the application as much of the requested data as possible, subject to the amount of data currently available in the stream's data buffer, and returns immediately.

For both blocking and non-blocking transfers, the function returns the amount of bytes that were actually read from the stream within the **`pdwBytesRead`** parameter.

Prototype

```
DWORD DLLCALLCONV WDU_StreamRead(
    HANDLE hStream,
    PVOID pBuffer,
    DWORD bytes,
    DWORD *pdwBytesRead);
```

Parameters

Name	Type	Input/Output
hStream	WDU_STREAM_HANDLE	Input
pBuffer	PVOID	Output
bytes	DWORD	Input
pdwBytesRead	DWORD*	Output

Description

Name	Description
hStream	A unique identifier for the stream, as returned by `WDU_StreamOpen`()
pBuffer	Pointer to a data buffer to be filled with the data read from the stream
bytes	Number of bytes to read from the stream
pdwBytesRead	Pointer to a value indicating the number of bytes actually read from the stream

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or `WD_OPERATION_FAILED` on failure.
In case of failure, call `WDU_StreamGetStatus`() [B.4.9.6] to get the current stream status.

B.4.9.4. WDU_StreamWrite

Purpose

Writes data from the applciation to a write stream.

For a blocking stream (`fBlocking=TRUE` — see `WDU_StreamOpen`()), the call to this function is blocked until the entire data is written to the stream, or until the stream's attempt to write to the device times out (i.e., the timeout period for transfers between the stream and the device, as set in the `dwRxTxTimeout` `WDU_StreamOpen`() parameter [B.4.9.1], expires).

For a non-blocking stream (`fBlocking=FALSE`), the function writes as much data as currently possible to the stream's data buffer, and returns immediately.

For both blocking and non-blocking transfers, the function returns the amount of bytes that were actually written to the stream within the **`pdwBytesWritten`** parameter.

Prototype

```
DWORD DLLCALLCONV WDU_StreamWrite(
    HANDLE hStream,
    const PVOID pBuffer,
    DWORD bytes,
    DWORD *pdwBytesWritten);
```

Parameters

Name	Type	Input/Output
hStream	WDU_STREAM_HANDLE	Input
pBuffer	const PVOID	Input
bytes	DWORD	Input
pdwBytesWritten	DWORD*	Output

Description

Name	Description
hStream	A unique identifier for the stream, as returned by `WDU_StreamOpen()`
pBuffer	Pointer to a data buffer containing the data to write to the stream
bytes	Number of bytes to write to the stream
pdwBytesWritten	Pointer to a value indicating the number of bytes actually written to the stream

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or `WD_OPERATION_FAILED` on failure.
In case of failure, call `WDU_StreamGetStatus()` [B.4.9.6] to get the current stream status.

B.4.9.5. WDU_StreamFlush

Purpose

Flushes a write stream, i.e., writes the entire contents of the stream's data buffer to the device. The function blocks until the completion of all pending I/O on the stream.

This function can be called for both blocking and non-blocking streams.

Prototype

```
DWORD DLLCALLCONV WDU_StreamFlush(
    WDU_STREAM_HANDLE hStream);
```

Parameters

Name	Type	Input/Output
hStream	WDU_STREAM_HANDLE	Input

Description

Name	Description
hStream	A unique identifier for the stream, as returned by `WDU_StreamOpen()`

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.9.6. WDU_StreamGetStatus

Purpose

Returns a stream's current status.

Prototype

```
DWORD DLLCALLCONV WDU_StreamGetStatus(
    WDU_STREAM_HANDLE hStream,
    BOOL *pfIsRunning,
    DWORD *pdwLastError,
    DWORD *pdwBytesInBuffer);
```

Parameters

Name	Type	Input/Output
hStream	WDU_STREAM_HANDLE	Input
pfIsRunning	BOOL*	Output
pdwLastError	DWORD*	Output
pdwBytesInBuffer	DWORD*	Output

Description

Name	Description
hStream	A unique identifier for the stream, as returned by `WDU_StreamOpen`()
pfIsRunning	Pointer to a value indicating the stream's current state: • `TRUE` — the stream is currently running • `FALSE` — the stream is currently stopped
pdwLastError	Pointer to the last error associated with the stream. Note: Calling the function also resets the stream's last error.
pdwBytesInBuffer	Pointer to the current bytes count in the stream's data buffer

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.9.7. WDU_StreamStop

Purpose

Stops an active stream, i.e., stops transfers between the stream and the device.
In the case of a write stream, the function flushes the stream — i.e., writes its contents to the device — before stopping it.

Prototype

```
DWORD DLLCALLCONV WDU_StreamStop(WDU_STREAM_HANDLE hStream);
```

Parameters

Name	Type	Input/Output
hStream	WDU_STREAM_HANDLE	Input

Description

Name	Description
hStream	A unique identifier for the stream, as returned by `WDU_StreamOpen()`

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.9.8. WDU_StreamClose

Purpose

Closes an open stream.
The function stops the stream, including flushing its data to the device (in the case of a write stream), before closing it.

Prototype

```
DWORD DLLCALLCONV WDU_StreamClose(WDU_STREAM_HANDLE hStream);
```

Parameters

Name	Type	Input/Output
hStream	WDU_STREAM_HANDLE	Input

Description

Name	Description
hStream	A unique identifier for the stream, as returned by `WDU_StreamOpen()`

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.10. WDU_ResetPipe

Purpose

Resets a pipe by clearing both the halt condition on the host side of the pipe and the stall condition on the endpoint. This function is applicable for all pipes except the control pipe (pipe 0).

Prototype

```
DWORD WDU_ResetPipe(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
dwPipeNum	DWORD	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface
dwPipeNum	The pipe's number

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

This function should be used if a pipe is halted, in order to clear the halt.

B.4.11. WDU_ResetDevice

Purpose

Resets a device.

Prototype

```
DWORD WDU_ResetDevice(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwOptions);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
dwOptions	DWORD	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface.
dwOptions	Can be either zero or: • **WD_USB_HARD_RESET** — reset the device even if it is not disabled. After using this option it is advised to set the interface device using WDU_SetInterface() [B.4.2]. • **WD_USB_CYCLE_PORT** — simulate unplugging and replugging of the device, prompting the operating system to re-enumerate the device without resetting it. This option is supported only on Windows XP and higher.

Return Value

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [B.8].

Remarks

- WDU_ResetDevice() is supported only on Windows and Windows CE, beginning with Windows CE 5.0.
  The **WD_USB_CYCLE_PORT** option is supported on Windows XP and higher.

- The function issues a request from the Windows USB driver to reset a hub port, provided the Windows USB driver supports this feature.

B.4.12. WDU_SelectiveSuspend

Purpose

Submits a request to suspend a given device (selective suspend), or cancels a previous suspend request.

Prototype

```
DWORD DLLCALLCONV WDU_SelectiveSuspend(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwOptions);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
dwOptions	DWORD	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface.
dwOptions	Can be set to either of the following `WDU_SELECTIVE_SUSPEND_OPTIONS` values: • **`WDU_SELECTIVE_SUSPEND_SUBMIT`** — submit a request to suspend the device. • **`WDU_SELECTIVE_SUSPEND_CANCEL`** — cancel a previous request to suspend the device.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8]. If the device is busy when a suspend request is submitted (**`dwOptions`**`=WDU_SELECTIVE_SUSPEND_SUBMIT`), the function returns `WD_OPERATION_FAILED`.

Remarks

`WDU_SelectiveSuspend`() is supported on Windows XP and higher.

B.4.13. WDU_Wakeup

Purpose

Enables/Disables the wakeup feature.

Prototype

```
DWORD WDU_Wakeup(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwOptions);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
dwOptions	DWORD	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface
dwOptions	Can be either: • **`WDU_WAKEUP_ENABLE`** — enable wakeup OR: • **`WDU_WAKEUP_DISABLE`** — disable wakeup

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.4.14. WDU_GetLangIDs

Purpose

Reads a list of supported language IDs and/or the number of supported language IDs from a device.

Prototype

```
DWORD DLLCALLCONV WDU_GetLangIDs(
    WDU_DEVICE_HANDLE hDevice,
    PBYTE pbNumSupportedLangIDs,
    WDU_LANGID *pLangIDs,
    BYTE bNumLangIDs);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
pbNumSupportedLangIDs	PBYTE	Output
pLangIDs	WDU_LANGID*	Output
bNumLangIDs	BYTE	Input

Description

Name	Description
hDevice	A unique identifier for the device/interface
pbNumSupportedLangIDs	Parameter to receive number of supported language IDs
pLangIDs	Array of language IDs. If **`bNumLangIDs`** is not zero the function will fill this array with the supported language IDs for the device.
bNumLangIDs	Number of IDs in the pLangIDs array

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

- If **dwNumLangIDs** is zero the function will return only the number of supported language IDs (in **pbNumSupportedLangIDs**) but will not update the language IDs array (**pLangIDs**) with the actual IDs. For this usage **pLangIDs** can be NULL (since it is not referenced) but **pbNumSupportedLangIDs** must not be NULL.

- **pbNumSupportedLangIDs** can be NULL if the user only wants to receive the list of supported language IDs and not the number of supported IDs.
  In this case **bNumLangIDs** cannot be zero and **pLangIDs** cannot be NULL.

- If the device does not support any language IDs the function will return success. The caller should therefore check the value of ***pbNumSupportedLangIDs** after the function returns.

- If the size of the **pLangIDs** array (**bNumLangIDs**) is smaller than the number of IDs supported by the device (***pbNumSupportedLangIDs**), the function will read and return only the first **bNumLangIDs** supported language IDs.

B.4.15. WDU_GetStringDesc

Purpose

Reads a string descriptor from a device by string index.

Prototype

```
DWORD DLLCALLCONV WDU_GetStringDesc(
    WDU_DEVICE_HANDLE hDevice,
    BYTE bStrIndex,
    PBYTE pbBuf,
    DWORD dwBufSize,
    WDU_LANGID langID,
    PDWORD pdwDescSize);
```

Parameters

Name	Type	Input/Output
hDevice	WDU_DEVICE_HANDLE	Input
bStrIndex	BYTE	Input
pbBuf	PBYTE	Output
dwBufSize	DWORD	Input
langID	WDU_LANGID	Input
pdwDescSize	PDWORD	Output

Description

Name	Description
hDevice	A unique identifier for the device/interface
bStrIndex	The index of the string descriptor to read
pbBuf	Pointer to a buffer to be filled with the string descriptor
dwBufSize	The size of the **`pbBuf`** buffer, in bytes
langID	The language ID to be used in the **get string descriptor** request. If this parameter is 0, the request will use the first supported language ID returned by the device.
pdwDescSize	An optional DWORD pointer to be filled with the size of the string descriptor read from the device. If `NULL`, the size of the string descriptor will not be returned.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

If the size of the **`pbBuf`** buffer is smaller than the size of the string descriptor (**`dwBufSize*pdwDescSize`**), the returned descriptor will be truncated to the provided buffer size (**`dwBufSize`**).

B.5. USB Data Types

The types described in this section are declared in the **WinDriver/include/windrvr.h** header file, unless otherwise specified in the documentation.

B.5.1. WD_DEVICE_REGISTRY_PROPERTY Enumeration

Enumeration of device registry property identifiers.
String properties are returned in NULL-terminated WCHAR array format.

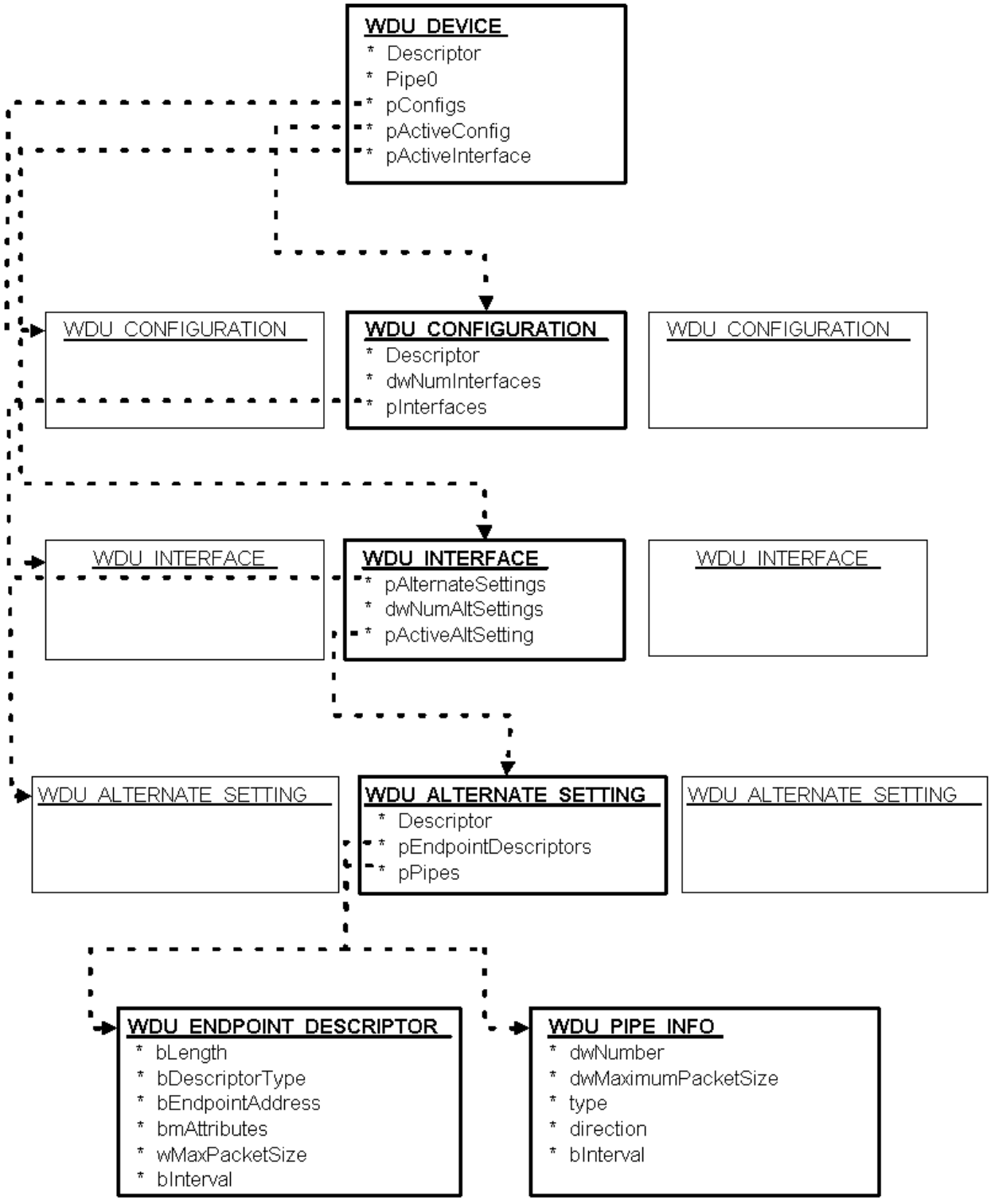
For more information regarding the properties described in this enumaration, refer to the description of the Windows `IoGetDeviceProperty`() function's `DeviceProperty` parameter in the Microsoft Development Network (MSDN) documentation.

Enum Value	Description
WdDevicePropertyDeviceDescription	Device description
WdDevicePropertyHardwareID	The device's hardware IDs
WdDevicePropertyCompatibleIDs	The device's compatible IDs
WdDevicePropertyBootConfiguration	The hardware resources assigned to the device by the firmware, in raw data form
WdDevicePropertyBootConfigurationTranslated	The hardware resources assigned to the device by the firmware, in translated form
WdDevicePropertyClassName	The name of the device's setup class, in text format
WdDevicePropertyClassGuid	The GUID for the device's setup class (string format)
WdDevicePropertyDriverKeyName	The name of the driver-specific registry key
WdDevicePropertyManufacturer	Device manufacturer string
WdDevicePropertyFriendlyName	Friendly device name (typically defined by the class installer), which can be used to distinguish between two similar devices
WdDevicePropertyLocationInformation	Information about the device's Location on the bus (string format). The interpertation of this information is bus-specific.
WdDevicePropertyPhysicalDeviceObjectName	The name of the Physical Device Object (PDO) for the device
WdDevicePropertyBusTypeGuid	The GUID for the bus to which the device is connected
WdDevicePropertyLegacyBusType	The bus type (e.g., PCIBus or PCMCIABus)
WdDevicePropertyBusNumber	The legacy bus number of the bus to which the device is connected
WdDevicePropertyEnumeratorName	The name of the device's enumerator (e.g., "PCI" or "root")
WdDevicePropertyAddress	The device's bus address. The interpertation of this address is bus-specific.
WdDevicePropertyUINumber	A number associated with the device that can be displayed in the user interface
WdDevicePropertyInstallState	The device's installation state
WdDevicePropertyRemovalPolicy	The device's current removal policy (Windows XP and later)

B.5.2. USB Structures

The following figure depicts the structure hierarchy used by WinDriver's USB API. The arrays situated in each level of the hierarchy may contain more elements than are depicted in the diagram. Arrows are used to represent pointers. In the interest of clarity, only one structure at each level of the hierarchy is depicted in full detail (with all of its elements listed and pointers from it pictured).

Figure B.2. WinDriver USB Structures



B.5.2.1. WDU_MATCH_TABLE Structure

USB match table structure.

(*) For all field members, if value is set to zero — match all.

Field	Type	Description
wVendorId	WORD	Required USB Vendor ID to detect, as assigned by USB-IF (*)
wProductId	WORD	Required USB Product ID to detect, as assigned by the product manufacturer (*)
bDeviceClass	BYTE	The device's class code, as assigned by USB-IF (*)
bDeviceSubClass	BYTE	The device's sub-class code, as assigned by USB-IF (*)
bInterfaceClass	BYTE	The interface's class code, as assigned by USB-IF (*)
bInterfaceSubClass	BYTE	The interface's sub-class code, as assigned by USB-IF (*)
bInterfaceProtocol	BYTE	The interface's protocol code, as assigned by USB-IF (*)

B.5.2.2. WDU_EVENT_TABLE Structure

USB events table structure.
This structure is defined in **WinDriver/include/wdu_lib.h**.

Field	Type	Description
pfDeviceAttach	WDU_ATTACH_CALLBACK	Will be called by WinDriver when a device is attached
pfDeviceDetach	WDU_DETACH_CALLBACK	Will be called by WinDriver when a device is detached
pfPowerChange	WDU_POWER_CHANGE_CALLBACK	Will be called by WinDriver when there is a change in a device's power state
pUserData	PVOID	Pointer to user-mode data to be passed to the callbacks

B.5.2.3. WDU_DEVICE Structure

USB device information structure.

Field	Type	Description
Descriptor	WDU_DEVICE_DESCRIPTOR	Device descriptor information structure [B.5.2.7]
Pipe0	WDU_PIPE_INFO	Pipe information structure [B.5.2.11] for the device's default control pipe (pipe 0)
pConfigs	WDU_CONFIGURATION*	Pointer to the beginning of an array of configuration information structures [B.5.2.4] describing the device's configurations
pActiveConfig	WDU_CONFIGURATION*	Pointer to the device's active configuration information structure [B.5.2.4], stored in the **`pConfigs`** array
pActiveInterface	WDU_INTERFACE* [WD_USB_MAX_INTERFACES]	Array of pointers to interface information structures [B.5.2.5]; the non-NULL elements in the array represent the device's active interfaces. On **Windows**, the number of active interfaces is the number of interfaces supported by the active configuration, as stored in the **`pActiveConfig->dwNumInterfaces`** field. On **Linux** and **Windows CE**, the number of active interfaces is currently always 1, because the `WDU_ATTACH_CALLBACK` device-attach callback [B.3.1] is called for each device interface.

B.5.2.4. WDU_CONFIGURATION Structure

Configuration information structure.

Field	Type	Description
Descriptor	WDU_CONFIGURATION_DESCRIPTOR	Configuration descriptor information structure [B.5.2.8]
dwNumInterfaces	DWORD	Number of interfaces supported by this configuration
pInterfaces	WDU_INTERFACE*	Pointer to the beginning of an array of interface information structures [B.5.2.5] for the configuration's interfaces

B.5.2.5. WDU_INTERFACE Structure

Interface information structure.

Field	Type	Description
pAlternateSettings	WDU_ALTERNATE_SETTING*	Pointer to the beginning of an array of alternate setting information structures [B.5.2.6] for the interface's alternate settings
dwNumAltSettings	DWORD	Number of alternate settings supported by this interface
pActiveAltSetting	WDU_ALTERNATE_SETTING*	Pointer to an alternate setting information structure [B.5.2.6] for the interface's active alternate setting

B.5.2.6. WDU_ALTERNATE_SETTING Structure

Alternate setting information structure.

Field	Type	Description
Descriptor	WDU_INTERFACE_DESCRIPTOR	Interface descriptor information structure [B.5.2.9]
pEndpointDescriptors	WDU_ENDPOINT_DESCRIPTOR*	Pointer to the beginning of an array of endpoint descriptor information structures [B.5.2.10] for the alternate setting's endpoints
pPipes	WDU_PIPE_INFO*	Pointer to the beginning of an array of pipe information structures [B.5.2.11] for the alternate setting's pipes

B.5.2.7. WDU_DEVICE_DESCRIPTOR Structure

USB device descriptor information structure.

Field	Type	Description
bLength	UCHAR	Size, in bytes, of the descriptor (18 bytes)
bDescriptorType	UCHAR	Device descriptor (0x01)
bcdUSB	USHORT	Number of the USB specification with which the device complies
bDeviceClass	UCHAR	The device's class
bDeviceSubClass	UCHAR	The device's sub-class
bDeviceProtocol	UCHAR	The device's protocol
bMaxPacketSize0	UCHAR	Maximum size of transferred packets
idVendor	USHORT	Vendor ID, as assigned by USB-IF
idProduct	USHORT	Product ID, as assigned by the product manufacturer
bcdDevice	USHORT	Device release number
iManufacturer	UCHAR	Index of manufacturer string descriptor
iProduct	UCHAR	Index of product string descriptor
iSerialNumber	UCHAR	Index of serial number string descriptor
bNumConfigurations	UCHAR	Number of possible configurations

B.5.2.8. WDU_CONFIGURATION_DESCRIPTOR Structure

USB configuration descriptor information structure.

Field	Type	Description
bLength	UCHAR	Size, in bytes, of the descriptor
bDescriptorType	UCHAR	Configuration descriptor (0x02)
wTotalLength	USHORT	Total length, in bytes, of data returned
bNumInterfaces	UCHAR	Number of interfaces
bConfigurationValue	UCHAR	Configuration number
iConfiguration	UCHAR	Index of string descriptor that describes this configuration
bmAttributes	UCHAR	Power settings for this configuration: • `D6` — self-powered • `D5` — remote wakeup (allows device to wake up the host)
MaxPower	UCHAR	Maximum power consumption for this configuration, in `2mA` units

B.5.2.9. WDU_INTERFACE_DESCRIPTOR Structure

USB interface descriptor information structure.

Field	Type	Description
bLength	UCHAR	Size, in bytes, of the descriptor (9 bytes)
bDescriptorType	UCHAR	Interface descriptor (0x04)
bInterfaceNumber	UCHAR	Interface number
bAlternateSetting	UCHAR	Alternate setting number
bNumEndpoints	UCHAR	Number of endpoints used by this interface
bInterfaceClass	UCHAR	The interface's class code, as assigned by USB-IF
bInterfaceSubClass	UCHAR	The interface's sub-class code, as assigned by USB-IF
bInterfaceProtocol	UCHAR	The interface's protocol code, as assigned by USB-IF
iInterface	UCHAR	Index of string descriptor that describes this interface

B.5.2.10. WDU_ENDPOINT_DESCRIPTOR Structure

USB endpoint descriptor information structure.

Field	Type	Description
bLength	UCHAR	Size, in bytes, of the descriptor (7 bytes)
bDescriptorType	UCHAR	Endpoint descriptor (0x05)
bEndpointAddress	UCHAR	Endpoint address: Use bits 0–3 for endpoint number, set bits 4–6 to zero (0), and set bit 7 to zero (0) for outbound data and to one (1) for inbound data (ignored for control endpoints).
bmAttributes	UCHAR	Specifies the transfer type for this endpoint (control, interrupt, isochronous or bulk). See the USB specification for further information.
wMaxPacketSize	USHORT	Maximum size of packets this endpoint can send or receive
bInterval	UCHAR	Interval, in frame counts, for polling endpoint data transfers. Ignored for bulk and control endpoints. Must equal 1 for isochronous endpoints. May range from 1 to 255 for interrupt endpoints.

B.5.2.11. WDU_PIPE_INFO Structure

USB pipe information structure.

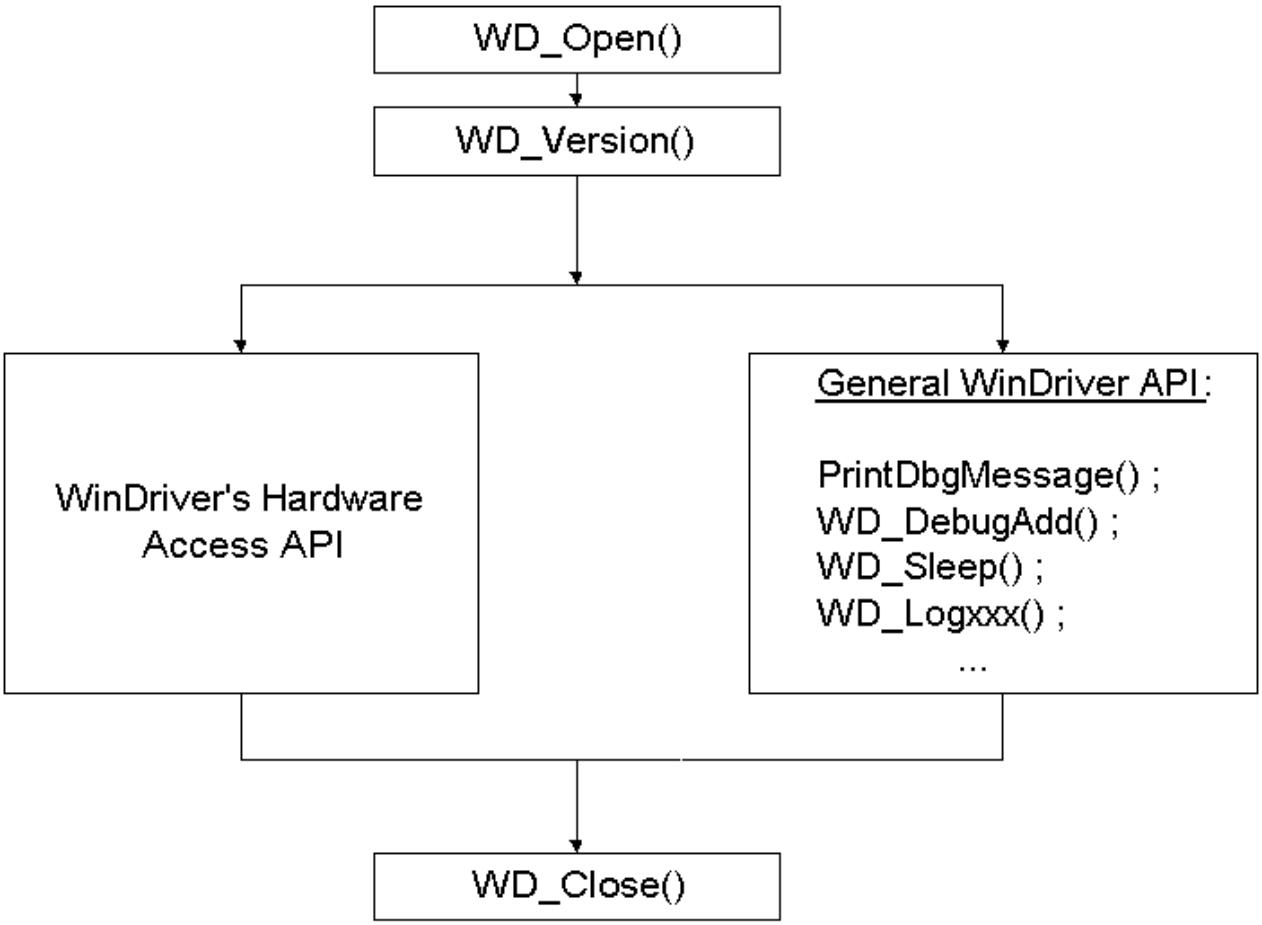
Field	Type	Description
dwNumber	DWORD	Pipe number; zero for the default control pipe
dwMaximumPacketSize	DWORD	Maximum size of packets that can be transferred using this pipe
type	DWORD	Transfer type for this pipe
direction	DWORD	Direction of the transfer: • **`WDU_DIR_IN`** or **`WDU_DIR_OUT`** for isochronous, bulk or interrupt pipes. • **`WDU_DIR_IN_OUT`** for control pipes.
dwInterval	DWORD	Interval in milliseconds. Relevant only to interrupt pipes.

B.6. General WD_xxx Functions

B.6.1. Calling Sequence WinDriver — General Use

The following is a typical calling sequence for the WinDriver API.

Figure B.3. WinDriver-API Calling Sequence



- We recommend calling the WinDriver function `WD_Version()` [B.6.3] after calling `WD_Open()` [B.6.2] and before calling any other WinDriver function. Its purpose is to return the WinDriver kernel module version number, thus providing the means to verify that your application is version compatible with the WinDriver kernel module.
- `WD_DebugAdd()` [B.6.6] and `WD_Sleep()` [B.6.8] can be called anywhere after `WD_Open()`

B.6.2. WD_Open()

Purpose

Opens a handle to access the WinDriver kernel module.
The handle is used by all WinDriver APIs, and therefore must be called before any other WinDriver API is called.

Prototype

```
HANDLE WD_Open(void);
```

Return Value

The handle to the WinDriver kernel module.
If device could not be opened, returns `INVALID_HANDLE_VALUE`.

Remarks

If you are a registered user, please refer to the documentation of `WD_License`() [B.6.9] for an example of how to register your WinDriver license.

Example

```
 HANDLE hWD;

hWD = WD_Open();
if (hWD == INVALID_HANDLE_VALUE)
{
    printf("Cannot open WinDriver device\n");
}
```

B.6.3. WD_Version()

Purpose

Returns the version number of the WinDriver kernel module currently running.

Prototype

```
DWORD WD_Version(
    HANDLE hWD,
    WD_VERSION *pVer);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pVer	WD_VERSION*	
• dwVer	DWORD	Output
• cVer	CHAR[128]	Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [B.6.2]
pVer	Pointer to a WinDriver version information structure:
• dwVer	The version number
• cVer	Version information string. The version string's size is limited to 128 characters (including the NULL terminator character).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Example

```
WD_VERSION ver;

BZERO(ver);
WD_Version(hWD, &ver);
printf("%s\n", ver.cVer);
if (ver.dwVer < WD_VER)
{
    printf("Error - incorrect WinDriver version\n");
}
```

B.6.4. WD_Close()

Purpose

Closes the access to the WinDriver kernel module.

Prototype

```
void WD_Close(HANDLE hWD);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [B.6.2]

Return Value

None

Remarks

This function must be called when you finish using WinDriver kernel module.

Example

```
WD_Close(hWD);
```

B.6.5. WD_Debug()

Purpose

Sets debugging level for collecting debug messages.

Prototype

```
DWORD WD_Debug(
    HANDLE hWD,
    WD_DEBUG *pDebug);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pDebug	WD_DEBUG*	Input
• dwCmd	DWORD	Input
• dwLevel	DWORD	Input
• dwSection	DWORD	Input
• dwLevelMessageBox	DWORD	Input
• dwBufferSize	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [B.6.2]
pDebug	Pointer to a debug information structure:
• dwCmd	Debug command: Set filter, Clear buffer, etc. For more details please refer to `DEBUG_COMMAND` in **windrvr.h**.
• dwLevel	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the debugging level to collect: Error, Warning, Info, Trace. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
• dwSection	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the sections to collect: I/O, Memory, Interrupt, etc. Use `S_ALL` for all. For more details please refer to `DEBUG_SECTION` in **windrvr.h**.
• dwLevelMessageBox	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the debugging level to print in a message box. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
• dwBufferSize	Used for `dwCmd=DEBUG_SET_BUFFER`. The size of buffer in the kernel.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Example

```
WD_DEBUG dbg;

BZERO(dbg);
dbg.dwCmd = DEBUG_SET_FILTER;
dbg.dwLevel = D_ERROR;
dbg.dwSection = S_ALL;
dbg.dwLevelMessageBox = D_ERROR;

WD_Debug(hWD, &dbg);
```

B.6.6. WD_DebugAdd()

Purpose

Sends debug messages to the debug log. Used by the driver code.

Prototype

```
DWORD WD_DebugAdd(
    HANDLE hWD,
    WD_DEBUG_ADD *pData);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pData	WD_DEBUG_ADD*	
• dwLevel	DWORD	Input
• dwSection	DWORD	Input
• pcBuffer	CHAR[256]	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [B.6.2]
pData	Pointer to an additional debug information structure:
• dwLevel	Assigns the level in the Debug Monitor, in which the data will be declared. If `dwLevel` is zero, `D_ERROR` will be declared. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
• dwSection	Assigns the section in the Debug Monitor, in which the data will be declared. If `dwSection` is zero, `S_MISC` section will be declared. For more details please refer to `DEBUG_SECTION` in **windrvr.h**.
• pcBuffer	The string to copy into the message log.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Example

```
WD_DEBUG_ADD add;

BZERO(add);
add.dwLevel = D_WARN;
add.dwSection = S_MISC;
sprintf(add.pcBuffer, "This message will be displayed in "
    "the Debug Monitor\n");
WD_DebugAdd(hWD, &add);
```

B.6.7. WD_DebugDump()

Purpose

Retrieves debug messages buffer.

Prototype

```
DWORD WD_DebugDump(
    HANDLE hWD,
    WD_DEBUG_DUMP *pDebugDump);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pDebug	WD_DEBUG_DUMP*	Input
• pcBuffer	PCHAR	Input/Output
• dwSize	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [B.6.2]
pDebugDump	Pointer to a debug dump information structure:
• pcBuffer	Buffer to receive debug messages
dwSize	Size of buffer in bytes

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Example

```
char buffer[1024];
WD_DEBUG_DUMP dump;
dump.pcBuffer=buffer;
dump.dwSize = sizeof(buffer);
WD_DebugDump(hWD, &dump);
```

B.6.8. WD_Sleep()

Purpose

Delays execution for a specific duration of time.

Prototype

```
DWORD WD_Sleep(
    HANDLE hWD,
    WD_SLEEP *pSleep);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pSleep	WD_SLEEP*	
• dwMicroSeconds	DWORD	Input
• dwOptions	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [B.6.2]
pSleep	Pointer to a sleep information structure
• dwMicroSeconds	Sleep time in microseconds — 1/1,000,000 of a second
• dwOptions	A bit-mask, which can be set to either of the following values: • **`Zero (0)`** — Delay execution by consuming CPU cycles (busy sleep); this is the default. • **`SLEEP_NON_BUSY`** — Delay execution without consuming CPU resources (non-busy sleep). Note: The accuracy of non-busy sleep is machine-dependent and cannot be guaranteed for short sleep intervals (< 1 millisecond).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

Example usage: to access slow response hardware.

Example

```
WD_Sleep slp;

BZERO(slp);
slp.dwMicroSeconds = 200;
WD_Sleep(hWD, &slp);
```

B.6.9. WD_License()

Purpose

Transfers the license string to the WinDriver kernel module and returns information regarding the license type of the specified license string.

When using the `WDU` USB APIs [B.2] your WinDriver license registration is done via the call to `WDU_Init`() [B.4.1], so you do not need to call `WD_License`() directly from your code.

Prototype

```
DWORD WD_License(
    HANDLE hWD,
    WD_LICENSE *pLicense);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pLicense	WD_LICENSE*	
• cLicense	CHAR[]	Input
• dwLicense	DWORD	Output
• dwLicense2	DWORD	Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [B.6.2]
pLicense	Pointer to a WinDriver license information structure:
• cLicense	A buffer to contain the license string that is to be transferred to the WinDriver kernel module. If an empty string is transferred, then WinDriver kernel module returns the current license type to the parameter `dwLicense`.
• dwLicense	Returns the license type of the specified license string (`cLicnese`). The return value is a bit-mask of license flags, defined as an enum in **windrvr.h**. Zero signifies an invalid license string. Additional flags for determining the license type are returned in **`dwLicense2`**, if needed.
• dwLicense2	Returns additional flags for determining the license type, if `dwLicense` cannot hold all the relevant information (otherwise — zero)

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

When using a registered version, this function must be called before any other WinDriver API call, apart from `WD_Open()` [B.6.2], in order to register the license from the code.

Example

Example usage: Add registration routine to your application:

```
DWORD RegisterWinDriver()
{
    HANDLE hWD;
    WD_LICENSE lic;
    DWORD dwStatus = WD_INVALID_HANDLE;

    hWD = WD_Open();
    if (hWD!=INVALID_HANDLE_VALUE)
    {
        BZERO(lic);
        /* Replace the following string with your license string: */
        strcpy(lic.cLicense, "12345abcde12345.CompanyName");
        dwStatus = WD_License(hWD, &lic);
        WD_Close(hWD);
    }

    return dwStatus;
}
```

B.7. User-Mode Utility Functions

This section describes a number of user-mode utility functions you will find useful for implementing various tasks. These utility functions are multi-platform, implemented on all operating systems supported by WinDriver.

B.7.1. Stat2Str

Purpose

Retrieves the status string that corresponds to a status code.

Prototype

```
const char *Stat2Str(DWORD dwStatus);
```

Parameters

Name	Type	Input/Output
dwStatus	DWORD	Input

Description

Name	Description
• dwStatus	A numeric status code

Return Value

Returns the verbal status description (string) that corresponds to the specified numeric status code.

Remarks

See Section B.8 for a complete list of status codes and strings.

B.7.2. get_os_type

Purpose

Retrieves the type of the operating system.

Prototype

```
OS_TYPE get_os_type(void);
```

Return Value

Returns the type of the operating system.
If the operating system type is not detected, returns `OS_CAN_NOT_DETECT`.

B.7.3. ThreadStart

Purpose

Creates a thread.

Prototype

```
DWORD ThreadStart(
    HANDLE *phThread,
    HANDLER_FUNC pFunc,
    void *pData);
```

Parameters

Name	Type	Input/Output
phThread	HANDLE*	Output
pFunc	typedef void (*HANDLER_FUNC)(void *pData);	Input
pData	VOID*	Input

Description

Name	Description
phThread	Returns the handle to the created thread
pFunc	Starting address of the code that the new thread is to execute. (The handler's prototype — `HANDLER_FUNC` — is defined in **utils.h**.)
pData	Pointer to the data to be passed to the new thread

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.7.4. ThreadWait

Purpose

Waits for a thread to exit.

Prototype

```
void ThreadWait(HANDLE hThread);
```

Parameters

Name	Type	Input/Output
hThread	HANDLE	Input

Description

Name	Description
hThread	The handle to the thread whose completion is awaited

Return Value

None

B.7.5. OsEventCreate

Purpose

Creates an event object.

Prototype

```
DWORD OsEventCreate(HANDLE *phOsEvent);
```

Parameters

Name	Type	Input/Output
phOsEvent	HANDLE*	Output

Description

Name	Description
phOsEvent	The pointer to a variable that receives a handle to the newly created event object

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.7.6. OsEventClose

Purpose

Closes a handle to an event object.

Prototype

```
void OsEventClose(HANDLE hOsEvent);
```

Parameters

Name	Type	Input/Output
hOsEvent	HANDLE	Input

Description

Name	Description
hOsEvent	The handle to the event object to be closed

Return Value

None

B.7.7. OsEventWait

Purpose

Waits until a specified event object is in the signaled state or the time-out interval elapses.

Prototype

```
DWORD OsEventWait(
    HANDLE hOsEvent,
    DWORD dwSecTimeout);
```

Parameters

Name	Type	Input/Output
hOsEvent	HANDLE	Input
dwSecTimeout	DWORD	Input

Description

Name	Description
hOsEvent	The handle to the event object
dwSecTimeout	Time-out interval of the event, in seconds. For an infinite wait, set the timeout to `INFINITE`.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.7.8. OsEventSignal

Purpose

Sets the specified event object to the signaled state.

Prototype

```
DWORD OsEventSignal(HANDLE hOsEvent);
```

Parameters

Name	Type	Input/Output
hOsEvent	HANDLE	Input

Description

Name	Description
hOsEvent	The handle to the event object

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.7.9. OsEventReset

Purpose

Resets the specified event object to the non-signaled state.

Prototype

```
DWORD OsEventReset(HANDLE hOsEvent);
```

Parameters

Name	Type	Input/Output
hOsEvent	HANDLE	Input

Description

Name	Description
hOsEvent	The handle to the event object

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.7.10. OsMutexCreate

Purpose

Creates a mutex object.

Prototype

```
DWORD OsMutexCreate(HANDLE *phOsMutex);
```

Parameters

Name	Type	Input/Output
phOsMutex	HANDLE*	Output

Description

Name	Description
phOsMutex	The pointer to a variable that receives a handle to the newly created mutex object

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.7.11. OsMutexClose

Purpose

Closes a handle to a mutex object.

Prototype

```
void OsMutexClose(HANDLE hOsMutex);
```

Parameters

Name	Type	Input/Output
hOsMutex	HANDLE	Input

Description

Name	Description
hOsMutex	The handle to the mutex object to be closed

Return Value

None

B.7.12. OsMutexLock

Purpose

Locks the specified mutex object.

Prototype

```
DWORD OsMutexLock(HANDLE hOsMutex);
```

Parameters

Name	Type	Input/Output
hOsMutex	HANDLE	Input

Description

Name	Description
hOsMutex	The handle to the mutex object to be locked

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.7.13. OsMutexUnlock

Purpose

Releases (unlocks) a locked mutex object.

Prototype

```
DWORD OsMutexUnlock(HANDLE hOsMutex);
```

Parameters

Name	Type	Input/Output
hOsMutex	HANDLE	Input

Description

Name	Description
hOsMutex	The handle to the mutex object to be unlocked

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.7.14. PrintDbgMessage

Purpose

Sends debug messages to the Debug Monitor.

Prototype

```
void PrintDbgMessage(
    DWORD dwLevel,
    DWORD dwSection,
    const char *format
    [, argument]...);
```

Parameters

Name	Type	Input/Output
dwLevel	DWORD	Input
dwSection	DWORD	Input
format	const char*	Input
argument		Input

Description

Name	Description
dwLevel	Assigns the level in the Debug Monitor, in which the data will be declared. If zero, `D_ERROR` will be declared. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
dwSection	Assigns the section in the Debug Monitor, in which the data will be declared. If zero, `S_MISC` will be declared. For more details please refer to DEBUG_SECTION in **windrvr.h**.
format	Format-control string
argument	Optional arguments, limited to 256 bytes

Return Value

None

B.7.15. WD_LogStart

Purpose

Opens a log file.

Prototype

```
DWORD WD_LogStart(
    const char *sFileName,
    const char *sMode);
```

Parameters

Name	Type	Input/Output
sFileName	const char*	Input
sMode	const char*	Input

Description

Name	Description
sFileName	Name of log file to be opened
sMode	Type of access permitted. For example, NULL or **w** opens an empty file for writing, and if the given file exists, its contents are destroyed; **a** opens a file for writing at the end of the file (i.e., append).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

Remarks

Once a log file is opened, all API calls are logged in this file.
You may add your own printouts to the log file by calling `WD_LogAdd()` [B.7.17].

B.7.16. WD_LogStop

Purpose

Closes a log file.

Prototype

```
VOID WD_LogStop(void);
```

Return Value

None

B.7.17. WD_LogAdd

Purpose

Adds user printouts into log file.

Prototype

```
VOID DLLCALLCONV WD_LogAdd(
    const char *sFormat
    [, argument ]...);
```

Parameters

Name	Type	Input/Output
sFormat	const char*	Input
argument		Input

Description

Name	Description
sFormat	Format-control string
argument	Optional format arguments

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [B.8].

B.8. WinDriver Status Codes

B.8.1. Introduction

Most of the WinDriver functions return a status code, where zero (`WD_STATUS_SUCCESS`) means success and a non-zero value means failure.
The `Stat2Str`() functions can be used to retrieve the status description string for a given status code. The status codes and their descriptive strings are listed below.

B.8.2. Status Codes Returned by WinDriver

Status Code	Description
WD_STATUS_SUCCESS	Success
WD_STATUS_INVALID_WD_HANDLE	Invalid WinDriver handle
WD_WINDRIVER_STATUS_ERROR	Error
WD_INVALID_HANDLE	Invalid handle
WD_INVALID_PIPE_NUMBER	Invalid pipe number
WD_READ_WRITE_CONFLICT	Conflict between read and write operations
WD_ZERO_PACKET_SIZE	Packet size is zero
WD_INSUFFICIENT_RESOURCES	Insufficient resources
WD_UNKNOWN_PIPE_TYPE	Unknown pipe type
WD_SYSTEM_INTERNAL_ERROR	Internal system error
WD_DATA_MISMATCH	Data mismatch
WD_NO_LICENSE	No valid license
WD_NOT_IMPLEMENTED	Function not implemented
WD_FAILED_ENABLING_INTERRUPT	Failed enabling interrupt
WD_INTERRUPT_NOT_ENABLED	Interrupt not enabled
WD_RESOURCE_OVERLAP	Resource overlap
WD_DEVICE_NOT_FOUND	Device not found
WD_WRONG_UNIQUE_ID	Wrong unique ID
WD_OPERATION_ALREADY_DONE	Operation already done
WD_USB_DESCRIPTOR_ERROR	USB descriptor error
WD_SET_CONFIGURATION_FAILED	Set configuration operation failed
WD_CANT_OBTAIN_PDO	Cannot obtain PDO
WD_TIME_OUT_EXPIRED	Timeout expired
WD_IRP_CANCELED	IRP operation cancelled

Status Code	Description
WD_FAILED_USER_MAPPING	Failed to map in user space
WD_FAILED_KERNEL_MAPPING	Failed to map in kernel space
WD_NO_RESOURCES_ON_DEVICE	No resources on the device
WD_NO_EVENTS	No events
WD_INVALID_PARAMETER	Invalid parameter
WD_INCORRECT_VERSION	Incorrect WinDriver version installed
WD_TRY_AGAIN	Try again
WD_INVALID_IOCTL	Received an invalid IOCTL
WD_OPERATION_FAILED	Operation failed
WD_INVALID_32BIT_APP	Received an invalid 32-bit IOCTL
WD_TOO_MANY_HANDLES	No room to add handle
WD_NO_DEVICE_OBJECT	Driver not installed

B.8.3. Status Codes Returned by USBD

The following WinDriver status codes comply with USBD_XXX status codes returned by the USB stack drivers.

Status Code	Description
USBD Status Types	
WD_USBD_STATUS_SUCCESS	USBD: Success
WD_USBD_STATUS_PENDING	USBD: Operation pending
WD_USBD_STATUS_ERROR	USBD: Error
WD_USBD_STATUS_HALTED	USBD: Halted
USBD Status Codes (NOTE: The status codes consist of one of the status types above and an error code, i.e., 0xXYYYYYYYL, where X=status type and YYYYYYY=error code. The same error codes may also appear with one of the other status types as well.)	
HC (Host Controller) Status Codes (NOTE: These use the `WD_USBD_STATUS_HALTED` status type.)	
WD_USBD_STATUS_CRC	HC status: CRC
WD_USBD_STATUS_BTSTUFF	HC status: Bit stuffing
WD_USBD_STATUS_DATA_TOGGLE_MISMATCH	HC status: Data toggle mismatch
WD_USBD_STATUS_STALL_PID	HC status: PID stall
WD_USBD_STATUS_DEV_NOT_RESPONDING	HC status: Device not responding
WD_USBD_STATUS_PID_CHECK_FAILURE	HC status: PID check failed

Status Code	Description
WD_USBD_STATUS_UNEXPECTED_PID	HC status: Unexpected PID
WD_USBD_STATUS_DATA_OVERRUN	HC status: Data overrun
WD_USBD_STATUS_DATA_UNDERRUN	HC status: Data underrun
WD_USBD_STATUS_RESERVED1	HC status: Reserved1
WD_USBD_STATUS_RESERVED2	HC status: Reserved2
WD_USBD_STATUS_BUFFER_OVERRUN	HC status: Buffer overrun
WD_USBD_STATUS_BUFFER_UNDERRUN	HC status: Buffer Underrun
WD_USBD_STATUS_NOT_ACCESSED	HC status: Not accessed
WD_USBD_STATUS_FIFO	HC status: FIFO
For Windows only:	
WD_USBD_STATUS_XACT_ERROR	HC status: The host controller has set the Transaction Error (XactErr) bit in the transfer descriptor's status field
WD_USBD_STATUS_BABBLE_DETECTED	HC status: Babble detected
WD_USBD_STATUS_DATA_BUFFER_ERROR	HC status: Data buffer error
For Windows CE only:	
WD_USBD_STATUS_ISOCH	USBD: Isochronous transfer failed
WD_USBD_STATUS_NOT_COMPLETE	USBD: Transfer not completed
WD_USBD_STATUS_CLIENT_BUFFER	USBD: Cannot write to buffer
For all platforms:	
WD_USBD_STATUS_CANCELED	USBD: Transfer cancelled
Returned by HCD (Host Controller Driver) if a transfer is submitted to an endpoint that is stalled:	
WD_USBD_STATUS_ENDPOINT_HALTED	HCD: Transfer submitted to stalled endpoint
Software Status Codes (NOTE: Only the error bit is set):	
WD_USBD_STATUS_NO_MEMORY	USBD: Out of memory
WD_USBD_STATUS_INVALID_URB_FUNCTION	USBD: Invalid URB function
WD_USBD_STATUS_INVALID_PARAMETER	USBD: Invalid parameter
Returned if client driver attempts to close an endpoint, interface, or configuration with outstanding transfers:	
WD_USBD_STATUS_ERROR_BUSY	USBD: Attempted to close endpoint/interface/configuration with outstanding transfer

Status Code	Description
*Returned by USBD if it cannot complete a URB request. Typically this will be returned in the URB status field (when the IRP is completed) with a more specific error code. The IRP status codes are indicated in WinDriver's Debug Monitor tool (**wddebug_gui / wddebug**):*	
WD_USBD_STATUS_REQUEST_FAILED	USBD: URB request failed
WD_USBD_STATUS_INVALID_PIPE_HANDLE	USBD: Invalid pipe handle
Returned when there is not enough bandwidth available to open a requested endpoint:	
WD_USBD_STATUS_NO_BANDWIDTH	USBD: Not enough bandwidth for endpoint
Generic HC (Host Controller) error:	
WD_USBD_STATUS_INTERNAL_HC_ERROR	USBD: Host controller error
Returned when a short packet terminates the transfer, i.e., USBD_SHORT_TRANSFER_OK bit not set:	
WD_USBD_STATUS_ERROR_SHORT_TRANSFER	USBD: Transfer terminated with short packet
Returned if the requested start frame is not within `USBD_ISO_START_FRAME_RANGE` of the current USB frame (NOTE: The stall bit is set):	
WD_USBD_STATUS_BAD_START_FRAME	USBD: Start frame outside range
Returned by HCD (Host Controller Driver) if all packets in an isochronous transfer complete with an error:	
WD_USBD_STATUS_ISOCH_REQUEST_FAILED	HCD: Isochronous transfer completed with error
Returned by USBD if the frame length control for a given HC (Host Controller) is already taken by another driver:	
WD_USBD_STATUS_FRAME_CONTROL_OWNED	USBD: Frame length control already taken
Returned by USBD if the caller does not own frame length control and attempts to release or modify the HC frame length:	
WD_USBD_STATUS_FRAME_CONTROL_NOT_OWNED	USBD: Attempted operation on frame length control not owned by caller
Additional software error codes added for USB 2.0 (for Windows only):	
WD_USBD_STATUS_NOT_SUPPORTED	USBD: API not supported/ implemented
WD_USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR	USBD: Invalid configuration descriptor
WD_USBD_STATUS_INSUFFICIENT_RESOURCES	USBD: Insufficient resources
WD_USBD_STATUS_SET_CONFIG_FAILED	USBD: Set configuration failed
WD_USBD_STATUS_BUFFER_TOO_SMALL	USBD: Buffer too small

Status Code	Description
WD_USBD_STATUS_INTERFACE_NOT_FOUND	USBD: Interface not found
WD_USBD_STATUS_INAVLID_PIPE_FLAGS	USBD: Invalid pipe flags
WD_USBD_STATUS_TIMEOUT	USBD: Timeout
WD_USBD_STATUS_DEVICE_GONE	USBD: Device gone
WD_USBD_STATUS_STATUS_NOT_MAPPED	USBD: Status not mapped
Extended isochronous error codes returned by USBD These errors appear in the packet status field of an isochronous transfer:	
WD_USBD_STATUS_ISO_NOT_ACCESSED_BY_HW	USBD: The controller did not access the TD associated with this packet
WD_USBD_STATUS_ISO_TD_ERROR	USBD: Controller reported an error in the TD
WD_USBD_STATUS_ISO_NA_LATE_USBPORT	USBD: The packet was submitted in time by the client but failed to reach the miniport in time
WD_USBD_STATUS_ISO_NOT_ACCESSED_LATE	USBD: The packet was not sent because the client submitted it too late to transmit

Appendix C
Troubleshooting and Support

Please refer to the online WinDriver support page — http://www.jungo.com/st/support/windriver/ — for additional resources for developers, including

- Technical documents
- FAQs
- Samples
- Quick start guides

Appendix D
Evaluation Version Limitations

D.1. Windows WinDriver Evaluation Limitations

- Each time WinDriver is activated, an *Unregistered* message appears.
- When using DriverWizard, a dialogue box with a message stating that an evaluation version is being run appears on every interaction with the hardware.
- DriverWizard [5]:
  - Each time DriverWizard is activated, an *Unregistered* message appears.
  - An evaluation message is displayed on every interaction with the hardware using DriverWizard.
- WinDriver will function for only 30 days after the original installation.

D.2. Windows CE WinDriver Evaluation Limitations

- Each time WinDriver is activated, an *Unregistered* message appears.
- The WinDriver CE Kernel (**windrvr6.dll**) will operate for no more than 60 minutes at a time.
- DriverWizard [5] (used on a host Windows PC) —
  - Each time DriverWizard is activated, an *Unregistered* message appears.
  - An evaluation message is displayed on every interaction with the hardware using DriverWizard.

D.3. Linux WinDriver Evaluation Limitations

- Each time WinDriver is activated, an *Unregistered* message appears.
- DriverWizard [5]:
  - Each time DriverWizard is activated, an *Unregistered* message appears.
  - An evaluation message is displayed on every interaction with the hardware using DriverWizard.
- WinDriver's kernel module will work for no more than 60 minutes at a time. To continue working, the WinDriver kernel module must be reloaded (unload and load the module) using the following commands:

The following commands must be executed with root privileges.

To unload —

```
# /sbin/modprobe -r windrvr6
```

To load —

```
# <path to wdreg> windrvr6
```

wdreg is provided in the **WinDriver/util** directory.

Appendix E
Purchasing WinDriver

Visit the WinDriver order page on our web site — http://www.jungo.com/st/order_wd/ — to select your WinDriver product(s) and receive a quote. Then fill in the WinDriver order form — available for download from the order page — and send it to Jungo by email or fax (see details in the order form and in the online order page). If you have installed the evaluation version of WinDriver, you can also find the order form in the **WinDriver/docs** directory, or access it via **Start | WinDriver | Order Form** on Windows.

The WinDriver license string will be emailed to you immediately.
Your WinDriver package will be sent to you via courier or registered mail.

Feel free to contact us with any question you may have. For full contact information, visit our contact web page: http://www.jungo.com/st/company/contact-us/.

Appendix F
Distributing Your Driver — Legal Issues

WinDriver is licensed per-seat. The WinDriver license allows one developer on a single computer to develop an unlimited number of device drivers, and to freely distribute the created drivers without royalties, as outlined in the license agreement in the **WinDriver/docs/wd_license.pdf** file.

Appendix G
Additional Documentation

Updated Manuals

The most updated WinDriver user manuals can be found on Jungo's site at http://www.jungo.com/st/support/windriver/.

Version History

If you wish to view WinDriver version history, refer to the WinDriver release notes, available online at http://www.jungo.com/st/support/windriver/wdver/. The release notes include a list of the new features, enhancements and fixes that have been added in each WinDriver version.

Technical Documents

For additional information, refer to the WinDriver Technical Documents database: http://www.jungo.com/st/support/tech_docs_indexes/main_index.html.
This database includes detailed descriptions of WinDriver's features, utilities and APIs and their correct usage, troubleshooting of common problems, useful tips and answers to frequently asked questions.